4-2024

Greening large language models of code

Jieke SHI

Zhou YANG

Hong Jin KANG

Bowen XU

Junda HE

See next page for additional authors

Citation

SHI, Jieke; YANG, Zhou; KANG, Hong Jin; XU, Bowen; HE, Junda; and LO, David. Greening large language models of code. (2024). *ICSE-SEIS'24: Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society, Lisbon, Portugal, April 14-20*. 142-153.
Available at: https://ink.library.smu.edu.sg/sis_research/9249

Greening Large Language Models of Code

Jieke Shi◇, Zhou Yang◇, Hong Jin Kang♠, Bowen Xu♣, Junda He◇, and David Lo◇
◇School of Computing and Information Systems, Singapore Management University, Singapore
♠Department of Computer Science, University of California, Los Angeles, USA
♣Department of Computer Science, North Carolina State University, Raleigh, USA
{jiekeshi, zyang, jundahe, davidlo}@smu.edu.sg, hjkang@cs.ucla.edu, bxu22@ncsu.edu

ABSTRACT

Large language models of code have shown remarkable effectiveness across various software engineering tasks. Despite the availability of many cloud services built upon these powerful models, there remain several scenarios where developers cannot take full advantage of them, stemming from factors such as restricted or unreliable internet access, institutional privacy policies that prohibit external transmission of code to third-party vendors, and more. Therefore, developing a compact, efficient, and yet energy-saving model for deployment on developers' devices becomes essential.

To this aim, we propose Avatar, a novel approach that crafts a deployable model from a large language model of code by optimizing it in terms of model size, inference latency, energy consumption, and carbon footprint while maintaining a comparable level of effectiveness (e.g., prediction accuracy on downstream tasks). The key idea of Avatar is to formulate the optimization of language models as a multi-objective configuration tuning problem and solve it with the help of a Satisfiability Modulo Theories (SMT) solver and a tailored optimization algorithm. The SMT solver is used to form an appropriate configuration space, while the optimization algorithm identifies the Pareto-optimal set of configurations for training the optimized models using knowledge distillation. We evaluate Avatar with two popular language models of code, i.e., CodeBERT and GraphCodeBERT, on two popular tasks, i.e., vulnerability prediction and clone detection. We use Avatar to produce optimized models with a small size (3 MB), which is 160× smaller than the original large models. On the two tasks, the optimized models significantly reduce the energy consumption (up to 184× less), carbon footprint (up to 157× less), and inference latency (up to 76× faster), with only a negligible loss in effectiveness (1.67%).

KEYWORDS

Language Models of Code, Configuration Tuning, Multi-Objective Optimization

ACM Reference Format:
Jieke Shi◇, Zhou Yang◇, Hong Jin Kang♠, Bowen Xu♣, Junda He◇, and David Lo◇. 2024. Greening Large Language Models of Code. In *Software Engineering in Society (ICSE-SEIS'24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3639475.3640097

†Zhou Yang is the corresponding author.

LAY ABSTRACT

Large language models of code have proven to be highly effective for various software engineering tasks, such as spotting program defects and helping developers write code. While many cloud services built on these models (e.g., GitHub Copilot) are now accessible, several factors, such as unreliable internet access (e.g., over 20% of GitHub Copilot's issues are related to network connectivity [22]) and privacy concerns (e.g., Apple has banned the internal use of external AI tools to protect confidential data [53]), hinder developers from fully utilizing these services. Therefore, deploying language models of code on developers' devices like laptops appears promising. However, local deployment faces challenges: (1) Consumer-grade personal devices typically lack sufficient memory and the high-performance CPUs/GPUs required for efficient model execution; (2) Even if the hardware requirements are met, deploying the models on many devices can result in considerable energy consumption and carbon emissions, negatively impacting environmental sustainability.

To address these challenges, we present Avatar, an innovative approach that optimizes large language models of code and enables their deployment on consumer-grade devices. Avatar can optimize two popular models from a large size of 481 MB to a compact size of 3 MB, resulting in significant reductions in inference time, energy consumption, and carbon emissions by hundreds of times. Our technique effectively lowers the entry barrier for leveraging large language models of code, making them available to ordinary developers without the need for high-performance computing equipment. Furthermore, it also contributes to a more sustainable and user-friendly software development environment.

1 INTRODUCTION

Recent years have seen a remarkable surge in Artificial Intelligence (AI)-powered services for software engineering, such as GitHub Copilot [23] and GitLab Auto DevOps [12]. This surge has brought a new level of automation to the software development process, significantly improving developer's productivity and the quality of software products. According to an economic analysis report released by GitHub, AI-powered services for software development could boost the global GDP by over $1.5 trillion by 2030 [13].

The foundation of these AI-powered services lies in large language models of code [35, 49, 55, 56, 82]. These models have shown superior performance in various software engineering tasks such as vulnerability detection [7, 33] and code completion [9, 47]. However, the services that utilize language models of code are typically hosted in the cloud, giving rise to several issues such as data leakage concerns [36, 48, 57, 80] and poor user experience due to network fluctuations [22]. Therefore, there is a growing need for deploying these models within the integrated development environments (IDEs) on developers' local machines. However, recent studies [65, 75] have highlighted several challenges associated with deploying language models of code, including their large size, long inference latency, high energy consumption, and considerable carbon footprint.

Typically, language models of code are large-sized with numerous parameters. For example, CodeBERT [18] and GraphCodeBERT [26], two popular language models of code, both have 125 million parameters, resulting in a file size of about 500 megabytes (MB). The recently released Code Llama model is even larger at over 130 gigabytes (GB) [58]. However, real-world deployment experiences, as observed by the Visual Studio team in deploying IDEs, have emphasized a preference for compact models, which are typically around 3 MB in size and can seamlessly function as IDE components or editor plug-ins even on low-end hardware devices [70]. Meanwhile, language models perform billions of floating-point operations (FLOPs) during inference. These massive computations cause long inference latency, often taking over 1.5 seconds to return a prediction [65]. Such delays can disrupt developers' workflow, ultimately resulting in a suboptimal user experience. Previous studies [4, 70] suggest that for a model deployed in IDEs to offer developers instantaneous assistance, its inference latency should ideally be within a few tens of milliseconds at most. The inability of language models of code to meet the above requirements gives rise to usability issues, consequently impeding their widespread deployment within developers' IDEs.

Furthermore, and perhaps even more importantly, the billions of FLOPs during inference entail significant energy consumption and carbon footprint, raising concerns about environmental and climate sustainability. Considering a CodeBERT deployed in IDEs, a developer typically needs to run it thousands of times per day, which is a common usage amount [31]. Such intensive usage results in an energy consumption of 0.32 kilowatt-hours (kWh), while a typical consumer-grade laptop has a battery capacity of around 70 watt-hours [40], i.e., 0.07 kWh. Consequently, a laptop's battery can only support a developer running CodeBERT for 0.22 hours, which is far from sufficient for a typical workday. This would frustrate developers and also hinder their ability to work flexibly in mobile environments. Moreover, the above energy cost of 0.32 kWh can translate into a considerable carbon footprint, amounting to approximately 0.14 kilograms of CO2 emissions. This carbon footprint is comparable to the emissions generated by driving a car for 0.6 miles.[1] With the expected widespread adoption of language models of code by many software developers in the near future, the cumulative carbon footprint stemming from model inference will become an increasingly pressing issue.

To date, few approaches have emerged to address the above issues [65, 75]. Shi et al. [65] propose Compressor, the state-of-the-art approach that can compress language models of code down to 3 MB and thereby improve their inference latency. Compressor adopts the knowledge distillation technique [34] to transfer knowledge from a large model to a tiny one with a well-crafted architecture searched by their proposed genetic algorithm. However, while Compressor excels at optimizing the model size and inference latency, it does not encompass the optimization of two other critical aspects, i.e., energy consumption and carbon footprint. Additionally, Compressor's search space for small model architectures is limited solely to hyperparameters related to model size, like the number of network layers. This limited scope excludes configurations that can significantly affect a model's effectiveness, like the choice of tokenizer [39]. Consequently, it falls short of identifying the optimal small model. These limitations necessitate our work. Our work still follows the idea of using knowledge distillation to optimize language models for the sake of size and inference latency, but offers a novel take on simultaneously addressing the issues of energy consumption and carbon footprint.

This paper proposes Avatar, a novel approach aimed at optimizing language models of code for real-world deployment. Avatar accomplishes this by formulating the seeking of an optimal model as a multi-objective configuration tuning problem, where the optimization objectives include the simultaneous minimization of model size, inference latency, energy consumption, and carbon footprint, while maintaining effectiveness (e.g., prediction accuracy) on downstream tasks.

Avatar starts by identifying the key configurations within language models that impact the above objectives. It then innovatively combines a Satisfiability Modulo Theories (SMT) solver with a tailored multi-objective optimization algorithm to solve the configuration tuning problem. The SMT solver is used to construct a configuration space that adheres to the 3 MB model size constraint, while the multi-objective optimization algorithm identifies the Pareto-optimal set of configurations, i.e., the set of configurations that cannot be improved in one objective without making sacrifices in another, thereby achieving the best trade-off among all objectives. To efficiently obtain the effectiveness of models during optimization without the need for expensive training and evaluation processes, Avatar builds a regression model serving as an effectiveness indicator. This indicator estimates a model's effectiveness solely based on its configurations, facilitating the quick identification of the Pareto-optimal configurations. Finally, Avatar leverages knowledge distillation to train a compact and environmentally-friendly model using the configurations from the Pareto-optimal set.

We evaluate Avatar using the same settings as the baseline method [65]. Our evaluation focuses on optimizing two representative language models of code: CodeBERT [18] and GraphCodeBERT [26]. We utilize two datasets for popular automated software engineering tasks: vulnerability prediction and clone detection. With Avatar, we produce optimized models with a compact size of 3 MB, a reduction of 160× compared to the original large language models. Across both tasks, these optimized models show a remarkable improvement in various aspects. They reduce inference latency by up to 76× compared to the original models, optimize energy consumption by up to 184× less, and reduce carbon footprint by up to 157× less. Importantly, these optimizations incur only a negligible loss in effectiveness, averaging 1.67%. Notably, Avatar outperforms the baseline method, Compressor, across all metrics. On average, Avatar achieves a 0.75% higher prediction accuracy. Additionally, it exhibits significant improvements in terms of inference latency (44× faster on average), energy consumption (up to 8× less), and

[1] All of these calculations on energy consumption and carbon footprint are based on the Machine Learning Emissions Calculator: https://mlco2.github.io/impact.

```
1  {
2    "tokenizer": "Byte-Pair Encoding",
3    "vocab_size": 50265,
4    "num_hidden_layers": 12,
5    "hidden_size": 768,
6    "hidden_act": "GELU",
7    "hidden_dropout_prob": 0.1,
8    "intermediate_size": 3072,
9    "num_attention_heads": 12,
10   "attention_probs_dropout_prob": 0.1,
11   "max_sequence_length": 512,
12   "position_embedding_type": "absolute",
13   "learning_rate": 1e-4,
14   "batch_size": 32
15 }
```

Listing 1: Typical tunable configurations of language models of code.

carbon footprint (up to 7× less). Moreover, we also highlight the benefits of Avatar in the context of cloud deployment, showing that the optimized models can process up to 9.7× more queries per second than the original large language models of code.

The contributions of this paper are summarized as follows:

- **Insight:** We are the first to propose optimizing language models of code in terms of the energy consumption and carbon footprint by tuning their configurations.
- **Technique:** We propose and implement Avatar, a novel approach that uses an SMT solver and a tailored multi-objective optimization algorithm to optimize language models of code in terms of model size, inference latency, energy consumption, and carbon footprint, while maintaining effectiveness.
- **Evaluation:** We perform a thorough evaluation of Avatar, and the results show that Avatar effectively optimizes language models of code, greatly outperforming the state-of-the-art approach.

2 PRELIMINARIES

Language Models of Code and Their Configurations. The recent development and adoption of language models of code have enabled state-of-the-art results to be achieved on code-related tasks [35, 49, 55, 56]. These powerful models are mainly built upon the Transformer architecture [74] and trained on large datasets of source code from various programming languages. Among these models, a notable category is encoder-only models such as CodeBERT [18] and GraphCodeBERT [26], which utilize solely the encoder component of Transformer and are specialized for program understanding tasks such as vulnerability detection [7] and code search [85]. These encoder-only models represent the software engineering community's early efforts at language models of code [35]. Due to their pioneering status, these models have long been used in various real-world applications like the Akvelon code search engine [2]. This has led to widespread popularity and social impact and thus motivated our study to focus on these models.

Typically, encoder-only language models of code have a number of configurations that can be tuned to achieve varying levels of model performance. Listing 1 shows an example of tunable configurations from the Hugging Face's implementation [15], with a total number of 13. Six of these configurations directly impact model size and inference latency, including the number of hidden layers, hidden size (i.e., the dimension of hidden layers), number of attention heads, vocabulary size, intermediate size (i.e., the dimension of feed-forward layers), and maximum sequence length. Larger values in these configurations tend to result in larger model sizes and longer inference latency, while smaller values may compromise model effectiveness (e.g., prediction accuracy). Compressor [65] focuses solely on tuning these configurations to optimize model size and inference latency at the cost of effectiveness.

However, there exist seven additional configurations that also contribute to model effectiveness. These include the choice of tokenizer, activation function for hidden layers, type of position embeddings, dropout rates for hidden layers and attention heads, learning rate, and batch size. For example, the choice of a tokenizer can affect a model's ability to capture the semantics of source code [39, 42, 64], thus impacting its overall effectiveness. In this study, we aim to tune all 13 configurations to achieve the best trade-off between model effectiveness and efficiency. We discuss the tuning space of these configurations and how to tune them in Section 3.

```
1  input $M$: language model of code (teacher model)
2  input $N$: small model (student model)
3  input $D$: training dataset
4  input $T$: temperature parameter
5  for $d$ in $D$:
6      $p, q = M(d), N(d)$
7      $loss = \text{softmax}(\frac{p}{T}) * \log\left(\text{softmax}(\frac{q}{T})\right) * T^2$
8      $N.\text{update}(loss)$
9  return $N$
```

Listing 2: Algorithm of knowledge distillation.

Knowledge Distillation. Knowledge distillation has proven to be an effective technique for optimizing large language models in terms of model size [41, 59, 65]. It compresses a large model (referred to as the teacher model) by training a small model (the student model) to mimic the behaviors of the large one (i.e., produces the same output given the same input) [5, 24, 34].

In line with recent work [65], our study leverages a task-specific distillation method introduced by Hinton et al. [34] to optimize language models of code. The algorithm of this method is shown in Listing 2. Specifically, given a language model of code that is fine-tuned for a specific task and a small model to be trained, we input training data into both models, collect the resulting output probability values (line 15), and then update the parameters of the small model (line 8) to minimize the training loss computed by the function shown in line 7. The intuition behind minimizing this loss function is to bring the outputs of the language and small models closer together. p_i and q_i in this function denote the outputs of the large and small models, respectively. T is the softmax function's temperature parameter, as Hinton et al. [34] introduced. Note that the language model producing p_i is fixed during the distillation process, while the small model producing q_i is trained.

Note that the above loss function does not necessitate ground-truth labels, only requiring the model's outputs. Thus, we follow Compressor [65] to use unlabeled data for training. This choice is

```
{
  "tokenizer": ["Byte-Pair Encoding", "WordPiece",
    ↪ "Unigram", "Word"],
  "vocab_size": range(1000, 50265),
  "num_hidden_layers": range(1, 12),
  "hidden_size": range(16, 768),
  "hidden_act": ["GELU", "ReLU", "SiLU", "GELU_new"],
  "hidden_dropout_prob": [0.1, 0.2, 0.3, 0.4, 0.5],
  "intermediate_size": range(16, 3072),
  "num_attention_heads": range(1, 12),
  "attention_probs_dropout_prob": [0.1, 0.2, 0.3, 0.4,
    ↪0.5],
  "max_sequence_length": range(256, 512),
  "position_embedding_type":["absolute", "relative_key",
    ↪ "relative_key_query"],
  "learning_rate": [1e-3, 1e-4, 5e-5],
  "batch_size": [16, 32, 64]
}
```

Listing 3: The configuration space of small models. It contains around 4.5×10^{19} plausible sets of configurations.

driven by the practical consideration that obtaining labeled data is typically costly and challenging, while ample unlabeled data can be readily collected from open-source software platforms like GitHub.

3 METHODOLOGY

3.1 Problem Formulation

As introduced in Section 1, we aim to optimize the model size, inference latency, energy consumption, and carbon footprint of language models of code while maintaining their effectiveness (e.g., prediction accuracy on downstream tasks). Among these objectives, the inference latency, energy consumption, and carbon footprint are all related to the model's computational cost during inference. We use floating-point operations (FLOPs) to measure computational cost, following prior studies [29, 61, 65]. FLOPs count how many multiply and accumulate operations the model performs for each prediction. The more FLOPs a model has, the more time it will take to make a prediction, the more energy it will consume, and the more CO_2 it will emit [61]. Therefore, we use FLOPs as the proxy for these three objectives. Then, combined with the model size and effectiveness, we formulate our optimization problem as follows:

$$\begin{aligned} \min_{c} \quad & \{\texttt{size}(c), \texttt{FLOPs}(c), -\texttt{effectiveness}(c)\} \\ \text{s.t.} \quad & c \in C \end{aligned} \tag{1}$$

where c is a set of configurations, and C defines the configuration space, as illustrated in Listing 3. Most of these configurations offer a range of adjustable integer or decimal values. For instance, the vocabulary size is adjustable to any integer value ranging from 1,000 to 50,265. Some others involve selecting from predefined options. The tokenizer requires a choice among four popular tokenization methods: Byte-Pair Encoding [62], WordPiece [76], Unigram [45], and Word [42]. Additionally, we set the hidden activation function and position embedding type as tunable configurations following the Hugging Face's implementation [15], which includes a few more advanced options than the original implementation of language models. The hidden activation function requires a choice

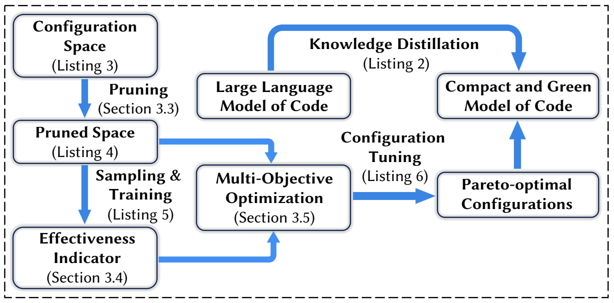
Configuration Space (Listing 3)
Pruning (Section 3.3)
Pruned Space (Listing 4)
Sampling & Training (Listing 5)
Effectiveness Indicator (Section 3.4)
Knowledge Distillation (Listing 2)
Large Language Model of Code
Compact and Green Model of Code
Configuration Tuning (Listing 6)
Multi-Objective Optimization (Section 3.5)
Pareto-optimal Configurations

Figure 1: The workflow of Avatar.

from four options: Gaussian Error Linear Unit (GELU) [32], Rectified Linear Unit (ReLU) [30], Sigmoid Linear Unit (SiLU) [14], and a new GELU implementation (GELU_new) [15]. The position embedding type offers three choices: absolute, relative_key [63], and relative_key_query [37]. In total, the configuration space contains about 4.5×10^{19} possible sets of configurations, which is much larger than the one used by Compressor that only tunes 5 configurations. Our configuration space is also extensible to include more configurations or more options for existing configurations, such as more tokenizer choices. Here we focus on the configuration space shown in Listing 3 as studies [39, 65] and Hugging Face's implementation [15] have explicitly shown that these configurations and options have a significant impact on model effectiveness.

Solving the problem posed by Equation 1 is challenging for three reasons: (1) the tuning space of configurations is quite huge, which makes brute force impractical since evaluating all configurations is computationally infeasible; (2) utilizing off-the-shelf Satisfiability Modulo Theories (SMT) solvers that support solving constrained optimization problems is not a viable approach for solving this problem. This is because obtaining model effectiveness necessitates training and testing the model. Such a process cannot be formulated as a mathematical function of configurations that SMT solvers can handle; (3) this multi-objective optimization problem comes with objectives that conflict with others. For example, a larger model typically has better effectiveness on downstream tasks but incurs higher FLOPs. Thus, solving Equation 1 involves finding a Pareto-optimal solution set, i.e., a set of trade-off solutions where no solution can be improved in one objective without degrading other objectives [10], rather than finding a single, unique solution.

3.2 Approach Overview

Pursuant to the above challenges, our approach, Avatar, is designed to solve the problem through a multi-step process outlined in Figure 1. First, we prune the configuration space using an SMT solver, with the 3 MB model size constraint suggested by prior studies [65, 70] as the pruning criterion (Section 3.3). This initial step removes configurations that are irrelevant to our objectives, thereby facilitating the subsequent identification of Pareto-optimal configurations. Next, we sample a small number of configurations from the pruned space and use them to train a regression model that can predict the effectiveness of a model initialized by a given set of configurations, i.e., build an effectiveness indicator (Section 3.4). Subsequently, we use a multi-objective optimization algorithm,

```
{
  "tokenizer": ["Byte-Pair Encoding", "WordPiece",
    ↪ "Unigram", "Word"],
  "vocab_size": range(1000, 46000),
  "num_hidden_layers": range(1, 12),
  "hidden_size": range(16, 256),
  "hidden_act": ["GELU", "ReLU", "SiLU", "GELU_new"],
  "hidden_dropout_prob": [0.1, 0.2, 0.3, 0.4, 0.5],
  "intermediate_size": range(32, 3072),
  "num_attention_heads": range(1, 12),
  "attention_probs_dropout_prob": [0.1, 0.2, 0.3, 0.4,
    ↪0.5],
  "max_sequence_length": range(256, 512),
  "position_embedding_type":["absolute", "relative_key",
    ↪ "relative_key_query"],
  "learning_rate": [1e-3, 1e-4, 5e-5],
  "batch_size": [16, 32, 64]
}
```

Listing 4: The pruned configuration space. It contains around 1.3×10^{19} **sets of configurations,** 28.9% **of the original one. The underlined entries are pruned (Section 3.3).**

assisted by the effectiveness indicator, to identify the set of Pareto-optimal configurations within the pruned space (Section 3.5). Finally, we train a compact and environmentally-friendly model with the configurations from the Pareto-optimal set using the knowledge distillation technique that we have introduced in Section 2. We describe these steps in detail below.

3.3 Pruning Configuration Space

The predefined configuration space shown in Listing 3 is incredibly large, with quintillions of possible configuration sets. However, only a fraction of them adhere to the constraints outlined in Section 1. For example, setting the vocabulary size to its maximum value of 50,265 will result in a model size that exceeds the 3 MB constraint, even with all other configurations minimized. Such configurations are thus considered irrelevant to our objectives and should be omitted from the configuration space to facilitate the subsequent process of identifying Pareto-optimal configurations.

We prune the configuration space by formulating and solving a constraint satisfaction problem using Microsoft Z3 [11], a state-of-the-art SMT solver known for efficiently handling nonlinear constrained optimization problems [6, 21]. While Z3 cannot directly solve our primary optimization problem, it performs well at identifying and excluding configurations that violate specified constraints. One crucial constraint is related to model size, as introduced in Section 1, which specifies that the model size cannot exceed 3 MB. This constraint is only explicit one suggested by prior studies [65, 70] while acceptable standards for other objectives have not been empirically specified. We formulate the constraint satisfaction problem as follows, where C represents the configuration space, and c denotes a set of configurations:

$$\texttt{size}(c) \leq 3\text{ MB} \quad \text{s.t.} \quad c \in C \tag{2}$$

Solving this constraint satisfaction problem yields multiple sets of configurations that satisfy the model size constraint, which can then be merged to craft a new configuration space.

```
input $C$: pruned configuration space
input $M$: language model of code (teacher model)
input $D$: training dataset
input $V$: validation dataset
input $T$: temperature parameter
input $k$: number of sampled configurations
$c = \texttt{sample}(C, k)$, $e = \{\}$
for $i$ in $k$:
    $N_i = \texttt{initialize}(c_i)$
    $N_i = \texttt{knowledge-distillation}(M, N_i, D, T)$
    $e_i = \texttt{test}(N_i, V)$
return $\texttt{Bayesian-Ridge-Regression}(\{c, e\})$
```

Listing 5: Algorithm for building an effectiveness indicator.

As pointed out in Section 2, a language model typically offers a handful of tunable configurations that directly determine the model size. Let v denote the vocabulary size, l denote the number of hidden layers, h denote the hidden size, i denote the intermediate size, a denote the number of attention heads, and s denote the maximum sequence length. Then the model size can be calculated as follows:

$$\begin{aligned} \texttt{size}(c) = & \frac{4(v+s+3)h}{1024 \times 1024} && \texttt{\# embedding layer} \\ & + \frac{4(4h^2 + (9+2i)h + i)l}{1024 \times 1024} && \texttt{\# transformer layers} \\ & + \frac{2h^2 + 4h + 2}{1024 \times 1024} && \texttt{\# classifier layer} \end{aligned} \tag{3}$$

The above formula follows the official implementation of Compressor [65] to calculate the actual file size of a model in MB. It breaks down a language model of code into three components: the embedding, transformer, and classifier layers. By summing these components, the formula calculates the total model size. Note that this formula only considers the six configurations that directly affect model size, while excluding other configurations like the tokenizer from our constraint satisfaction problem-solving process.

We then use the above formula and the raw configuration space as inputs to Z3, to find the configurations for which the formula evaluates to a value less than 3 MB. Considering that solving with Z3 can slow down significantly when dealing with an overly large configuration space [21, 73], we run Z3 by partitioning the configuration space into several smaller subspaces and processing them in parallel. Taking the vocabulary size as an example, we can partition the original range of 1,000 to 50,265 into 50 subranges, i.e., 1,000 to 2,000, 2,000 to 3,000, etc. These 50 subranges are then combined with the tuning ranges of other configurations, forming 50 subspaces. Each subspace's constraint satisfaction problem is treated as an independent task and solved in parallel using separate Z3 threads. Once all tasks are completed, we aggregate the results to form a new, pruned configuration space, as shown in Listing 4. The underlined entries, i.e., the vocabulary size, hidden size, and intermediate size, have been pruned. This process significantly reduces the configuration space from 4.5×10^{19} to 1.3×10^{19}, which accounts for only 28.9% of the original space. Notably, the pruned configuration space still contains a broad and diverse range of configurations, providing sufficient space to identify Pareto-optimal solutions.

3.4 Effectiveness Indicator

When tuning configurations, assessing the effectiveness of a model that has a given set of configurations is essential to determine whether it qualifies as a Pareto-optimal solution. However, obtaining model effectiveness through training and testing is computationally expensive. Inspired by recent work in leveraging machine learning techniques to predict the runtime performance of software [20, 27, 28], we propose to construct a regression model as a proxy for the training and testing process. Specifically, the regression model builds a computationally efficient function that maps a model's configurations to its effectiveness, enabling us to estimate a model's effectiveness using only the provided configuration as input. Consequently, this approach eliminates the need for resource-intensive model training and testing. We consider this regression model as an effectiveness indicator.

We follow the procedures outlined in Listing 5 to develop an effectiveness indicator. First, we randomly sample a set of configurations from the pruned configuration space (line 7). Next, we utilize the knowledge distillation technique introduced in Section 2 to train a model for each of these sampled configurations (line 10). We then evaluate the effectiveness of these models on the validation dataset (line 11), which has a similar distribution to the test dataset, but remains distinct and is not used for training. Subsequently, we use the sampled configurations and the corresponding effectiveness values to train a regression model that serves as our effectiveness indicator (line 12). For this purpose, we employ Bayesian Ridge Regression (BRR) [72]. BRR is a statistical regression method that combines Bayesian principles [51] with linear regression techniques [67]. It trains regression models by minimizing the squared difference between predicted and actual target values. BRR is particularly valuable when dealing with limited data points, which is the case for our effectiveness indicator since we have only a few sampled configurations. Note that the regression model usually takes numbers as inputs, while some of our configurations are strings. For these configurations, we use their corresponding indices in the tuning range as inputs to the regression model. For example, the tokenizer has four options, so we use 0, 1, 2, and 3 to represent them.

3.5 Multi-Objective Configuration Tuning

With the pruned configuration space and effectiveness indicator, we are now ready to introduce our innovative multi-objective configuration tuning algorithm, which is specifically designed to identify the set of Pareto-optimal configurations in terms of size, FLOPs, and effectiveness for optimizing large language models of code.

As presented in Listing 6, our algorithm takes the pruned configuration space, the effectiveness indicator, and the number of generations as inputs. It starts by generating an initial population of configuration sets by an adaptive random initialization method (line 5). These configurations are then assessed in terms of the three objectives (line 6): the size and FLOPs are calculated with the implementation of COMPRESSOR [65], while the effectiveness indicator predicts the effectiveness. The algorithm maintains an archive to store the Pareto-optimal configurations (line 7). This archive is initialized as an empty set and is updated throughout the algorithm's execution. Subsequently, it enters an iterative loop that runs for a specified number of generations. At each iteration, the algorithm applies three operators, i.e., two-point crossover, boundary random mutation, and correction, to generate new offspring from the population (lines 9 to 11). These offspring are then evaluated, and the archive of Pareto-optimal configurations is updated accordingly (lines 12 to 13). The next generation of population is selected from the current population and the offspring by a tournament selection method (line 14). After the loop terminates, the algorithm returns the archive of Pareto-optimal configurations (line 15). The main operators and steps are described in detail below.

```
1  input $C$: pruned configuration space
2  input $I$: effectiveness indicator
3  input $g$: number of generations
4  input $p$: population size
5  $P$ = adaptive-random-initialization($C, p$)
6  $W$ = calculate-objectives($P, I$)
7  $A$ = update-archive($P, W, \emptyset$)
8  for $i = 0$ to $g$:
9      $Q$ = two-point-crossover($P$)
10     $Q$ = boundary-random-mutation($Q$)
11     $Q$ = correction($Q$)
12     $W$ = calculate-objectives($Q, I$)
13     $A$ = update-archive($Q, W, A$)
14     $P$ = tournament-selection($P \cup Q$)
15 return $A$
```

Listing 6: Algorithm of multi-objective configuration tuning.

Adaptive Random Initialization. We aim to assemble an initial population of highly diverse configuration sets, which can facilitate more efficient exploration of the configuration space. To achieve this, we employ adaptive random initialization [1, 50], an extension of naive random search that attempts to maximize the Euclidean distance between the selected configurations in the population. Concretely, this method first randomly selects a configuration set c from the configuration space. It then randomly selects another configuration set c' and compares the Euclidean distance between c and c' with the distance between c and the other configuration sets already present in the population. If the distance between c and c' exceeds those between c and other configuration sets, c' is added to the population. Otherwise, c' is discarded. This process continues until the population reaches the desired size. Importantly, when calculating the Euclidean distance, as when training the effectiveness indicator, we replace the configuration in the form of strings with its corresponding numerical index within the tuning range.

Two-Point Crossover. This operator, commonly used in metaheuristic algorithms such as genetic algorithms to solve optimization problems [44, 66], aims to combine two parent configurations to generate new offspring configurations. It begins by randomly selecting two parent configurations and two crossover points. Subsequently, it swaps the values of the two parent configurations between these two crossover points to create two offspring configurations. For instance, if the two parent configurations are denoted as c_1 and c_2, and the selected crossover points are p_1 and p_2, the resulting offspring configurations are computed as follows: $c_1[0 : p_1]+c_2[p_1 : p_2]+c_1[p_2 :]$ and $c_2[0 : p_1]+c_1[p_1 : p_2]+c_2[p_2 :]$. Here, $c_1[0 : p_1]$ represents the values of c_1 before p_1, and $c_1[p_2 :]$

represents the values of c_1 from p_2 to the end. The generated offspring configurations are then added to the population.

Boundary Random Mutation. This operator introduces random modifications to the values of a configuration set, resulting in a new offspring configuration. Following recent work utilizing genetic algorithms for optimization problems [65, 79], we employ the boundary random mutation operator to generate offspring configurations. The process begins by randomly selecting a configuration from the population. Subsequently, for each configuration value within this selected configuration, a mutation rate r is randomly chosen from the range of $[0, 1]$. If r falls below a predefined threshold, the selected configuration value is set to a random value within its tuning range, while ensuring that the modified solution remains within the feasible configuration space, i.e., the boundary. The resulting offspring configuration is then incorporated into the population.

Correction. The above crossover and mutation operators may produce invalid offspring configurations that are unusable for initializing models. For example, according to the implementation of Hugging Face [15], a model's hidden size must be divisible by the number of attention heads; otherwise, the model will fail to initialize due to dimension misalignment errors. To address such cases and rectify them, our tuning algorithm employs correction operators. When it encounters invalid offspring configurations, it discards their values and proceeds to randomly select new values until the offspring configuration becomes valid.

Tournament Selection. The selection operator plays a key role in constructing the next generation from the existing population and the newly generated offspring. Using the tournament selection method [17], a well-established technique in metaheuristic algorithms, a fixed number of configurations are randomly selected from the combined pool of the current population and offspring. Then, the Pareto-optimal ones are selected from these configurations and added to the next generation, ensuring that the most promising candidates are retained for the next iteration.

As mentioned above, the algorithm manages and continuously updates an archive of Pareto-optimal configurations throughout its execution. When evaluating a configuration set, the algorithm compares it with the configurations already present in the archive. If the evaluated configuration set is not dominated by any other configuration set in the archive, it secures its place within the archive. Additionally, if any configuration set in the archive is found to be dominated by the new configuration set, it will be excluded from the archive. This process ensures the archive contains only non-dominated configurations, i.e., Pareto-optimal solutions. The algorithm terminates when the specified number of generations is reached, at which point it returns the archive of Pareto-optimal configurations. We then select a configuration set from the archive to train a compact and green model using knowledge distillation.

4 EMPIRICAL EVALUATION

Our evaluation aims to answer the following research questions:

- **RQ1 (Effectiveness):** How effective is AVATAR in optimizing language models of code?
- **RQ2 (Comparison):** How does AVATAR compare to the state-of-the-art method in optimizing language models of code?

Table 1: Overview of datasets used in our experiments.

Dataset	Labeled/Unlabeled Val/Test	Language	Source
Devign [86]	10,927/10,927 2,732/2,732	C	FFmpeg Qemu
BigCloneBench [69]	45,051/45,051 4,000/4,000	Java	SourceForge Google Code

4.1 Experimental Setup

Tasks and Datasets. Following the evaluation settings in the prior work [65], we assess the performance of AVATAR on two popular software engineering tasks: vulnerability prediction and clone detection. Table 1 provides an overview of the datasets used in our experiments. These datasets encompass different programming languages and sizes, allowing for a thorough evaluation of AVATAR. More details on the tasks and datasets are provided below.

The vulnerability prediction task involves determining whether a given code snippet is vulnerable or not. Integrating vulnerability prediction models into an IDE can significantly assist developers in identifying critical program defects early, thus enhancing software quality and reducing maintenance costs. For our experiment, we utilize the Devign dataset [86], which was released by Zhou et al. It contains 27,318 functions from two popular open-source C libraries, i.e., FFmpeg and Qemu. The dataset was constructed by manually annotating whether these functions contain vulnerabilities or not. We first follow the CodeXGLUE [49] benchmark for dataset splitting, allocating 80% for training, 10% for validation, and 10% for testing. To facilitate knowledge distillation, which requires unlabeled data, we follow COMPRESSOR [65] to evenly divide the training set into two mutually exclusive halves. One half is used for fine-tuning the language models, while the other, with erased labels, serves to train the model with configurations generated by AVATAR.

The clone detection task aims to identify whether two given functions are code clones, assisting in recognizing redundant implementations of the same functionalities during software maintenance. For evaluating AVATAR's effectiveness in clone detection, we select the widely-used BigCloneBench dataset [69]. This dataset is collected by mining the clones of specific functionalities in 25,000 Java projects sourced from SourceForge and Google Code platform. It includes over 6,000,000 pairs of cloned Java methods, along with 260,000 non-clone pairs. We follow recent studies [65, 79] to randomly select 90,102 examples (i.e., 10% of the original training dataset) for training and reserve 4,000 for validation and testing. Then, we divide the training data into labeled and unlabeled portions of equal size, which are for fine-tuning large models and training optimized models, respectively.

Language Models of Code. To evaluate AVATAR, we follow Shi et al. [65] to use two popular encoding-only language models of code: CodeBERT [18] and GraphCodeBERT [26]. These two models share the same architecture and have been language on the CodeSearchNet dataset [38]. CodeBERT undergoes pre-training with two tasks: masked language modeling, which predicts masked tokens in input texts, and replaced token detection, which identifies whether a token in a given input has been replaced. GraphCodeBERT also uses masked language modeling, but also incorporates code graph

Table 2: Results of AVATAR and the original language models on the two tasks. "CB" and "GCB" denote CodeBERT and GraphCodeBERT, respectively. "ACC" is the prediction accuracy. "LAT" is the inference latency. "E" is the energy consumption. "CO_2" is the CO_2 emission, i.e., the carbon footprint.

Model	Vulnerability Prediction					Clone Detection				
	ACC (%)	LAT (ms)	E (kWh)	CO_2 (kg)	GFLOPs	ACC (%)	LAT (ms)	E (kWh)	CO_2 (kg)	GFLOPs
CB (481 MB)	61.82	1394	0.32	0.14	138.4	96.10	1963	0.65	0.28	138.4
CB-AVATAR (**3 MB**)	60.87 (-0.95)	**29 (48×)**	**0.006 (53×)**	**0.003 (47×)**	**0.64 (216×)**	93.69 (-2.41)	**19 (103×)**	**0.006 (108×)**	**0.003 (93×)**	**1.14 (121×)**
GCB (481 MB)	61.57	1139	0.26	0.11	138.4	96.85	1539	0.52	0.22	138.4
GCB-AVATAR (**3 MB**)	61.12 (-0.45)	**15 (76×)**	**0.005 (52×)**	**0.002 (55×)**	**0.67 (207×)**	94.00 (-2.85)	**10 (154×)**	**0.002 (260×)**	**0.001 (220×)**	**0.80 (173×)**
Average Loss/Gain	-0.70	**62×**	**53×**	**51×**	**212×**	-2.63	**129×**	**184×**	**157×**	**147×**

structure information by predicting masked nodes in data flow graphs during pre-training. After pre-training, both CodeBERT and GraphCodeBERT can be fine-tuned on downstream tasks, enabling them to achieve state-of-the-art performance [49, 56, 82].

To fine-tune CodeBERT, we use the hyperparameter settings from the CodeXGLUE benchmark [49]. In the case of GraphCodeBERT, we follow the hyperparameter settings described in the GraphCodeBERT paper [26]. All models deliver results comparable to those reported in the previous study [82].

Evaluation Metrics. After obtaining the model trained with configurations tuned by AVATAR, we compare it with the language model and the model generated by our baseline method, COMPRESSOR, using six metrics: effectiveness, model size, inference latency, energy consumption, carbon footprint, and Giga floating-point operations (GFLOPs). Effectiveness is evaluated by prediction accuracy on the two downstream tasks, following prior studies [65, 79]. Model size is quantified in megabytes (MB). For inference latency, which is measured in milliseconds (ms), we standardize experimental conditions by limiting all models to use only 8 CPU cores, simulating running on a typical consumer-grade laptop. The testing datasets are used to query the models, and the average inference latency is calculated for each data example. Note that we use a batch size of 1 to replicate real-world scenarios where models are deployed on laptops and only process a single input at a time.

To evaluate energy consumption and carbon footprint, we use the Machine Learning Emissions Calculator[2], developed by Lacoste et al. [46]. The tool requires the total running time of a model as input and outputs the energy consumption and carbon footprint, measured in kilowatt-hours (kWh) and kilograms (kg), respectively. We record the total running time of the models on the testing datasets as input to the tool, and consistent with our inference latency evaluation, we use a batch size of 1. Additionally, as mentioned in Section 3, GFLOPs are commonly used to quantify the computational cost of a model, which is closely related to energy consumption and carbon footprint. Thus, we also report GFLOPs to illustrate how AVATAR contributes to environmental sustainability by reducing the computational cost of language models of code.

Implementation. We run all experiments on an Ubuntu 18.04 server equipped with an Intel Xeon E5-2698 CPU, 504 GB of RAM, and 8 Tesla V100 GPUs. To prune the configuration space with Z3, we partition it into 25,600 subspaces and execute Z3 in parallel across 80 CPU cores. For training the effectiveness indicator, we sample 20 sets of configurations from the pruned configuration space. In the multi-objective tuning algorithm, we configure the population size to be 20, with 50 generations. The crossover and mutation rates were set to 0.6 and 0.1, respectively.

[2]https://mlco2.github.io/impact/#compute

4.2 Effectiveness of AVATAR (RQ1)

After obtaining the Pareto-optimal configurations using AVATAR, we select the configuration with a model size closest to 3 MB for training the optimized model. This results in a model that is approximately 160× smaller than the original language model of code for each task. Table 2 shows the experimental results comparing the optimized models with the original ones. On the two tasks, the optimized models exhibit an average decrease in accuracy of only 1.67% ($\approx (0.70\% + 2.63\%)/2$) compared to the original large models. This accuracy result illustrates that AVATAR significantly optimizes model size with only a negligible loss in effectiveness on downstream tasks. Furthermore, the inference latency of the optimized models sees a substantial reduction on both tasks, with an average reduction of 62× for vulnerability detection and 129× for clone detection. Prior research [52] has suggested that software practitioners are willing to accept a small sacrifice in effectiveness in exchange for a significant improvement in usability. Therefore, we consider the reduced accuracy of the optimized models to be acceptable in practical applications.

Table 2 also presents results of optimizing language models in terms of environmental sustainability. We employ the Machine Learning Emissions Calculator [46] to calculate the energy consumption and carbon footprint of the optimized models, comparing them to the original ones. Note that these results are calculated using a single NVIDIA Tesla V100 GPU and encompass the cost of running the entire testing dataset rather than a single query. On both tasks, the energy consumption of the optimized models sees a significant reduction, averaging 53× and 184× less, respectively. This reduction extends to a corresponding decrease in carbon footprint, ranging from 51× to 157× less. Additionally, we observe a notable reduction in GFLOPs for the optimized models, with an average reduction of 212× and 147× on the two tasks, respectively. These results underscore the sustainability benefits that the optimized models can offer in real-world deployments.

Answers to RQ1: AVATAR effectively optimizes language models of code in terms of model size (160× smaller), inference latency (up to 76× faster), energy consumption (up to 184× less), and carbon footprint (up to 157× less), with only a negligible loss in effectiveness (1.67% on average).

Table 3: Results of Avatar and Compressor on the two tasks. "CB" and "GCB" denote CodeBERT and GraphCodeBERT, respectively. "ACC" is the prediction accuracy. "LAT" is the inference latency. "E" is the energy consumption. "CO_2" is the CO_2 emission, i.e., the carbon footprint.

Model	Vulnerability Prediction					Clone Detection				
	ACC (%)	LAT (ms)	E (kWh)	CO_2 (kg)	GFLOPs	ACC (%)	LAT (ms)	E (kWh)	CO_2 (kg)	GFLOPs
CB-Compressor (3 MB)	59.11	521	0.012	0.006	2.25	95.40	601	0.02	0.01	2.25
CB-Avatar (3 MB)	**60.87 (+1.76)**	**29 (18×)**	**0.006 (2×)**	**0.003 (2×)**	**0.64 (4×)**	**93.69 (-1.71)**	**19 (32×)**	**0.006 (3×)**	**0.003 (3×)**	**1.14 (2×)**
GCB-Compressor (3 MB)	59.99	702	0.016	0.007	2.25	92.15	747	0.025	0.011	2.25
GCB-Avatar (3 MB)	**61.12 (+1.13)**	**15 (47×)**	**0.005 (3×)**	**0.002 (4×)**	**0.67 (3×)**	**94.00 (+1.85)**	**10 (75×)**	**0.002 (13×)**	**0.001 (11×)**	**0.80 (3×)**
Average Loss/Gain	**+1.45**	**33×**	**3×**	**4×**	**4×**	**+0.07**	**54×**	**8×**	**7×**	**3×**

4.3 Avatar vs. Compressor (RQ2)

As the baseline for our experiments, we employ the approach, Compressor, proposed by Shi et al. [65]. To ensure a fair comparison, we directly utilize the models available in the official repository of Compressor. The models produced using Compressor and Avatar have a similar size at 3 MB. The evaluation results comparing these approaches are presented in Table 3.

Compared to the models optimized by Compressor, the models produced by Avatar exhibit a slightly higher accuracy, with an average improvement of 0.75% ($\approx (1.45\% + 0.07\%)/2$) on the two tasks. These results suggest that Avatar can optimize language models of code more effectively without compromising effectiveness as much as Compressor. More importantly, the models optimized by Avatar demonstrate significant improvements in inference latency on both tasks. Compressor produces models with an inference latency in the hundreds of milliseconds range, while the optimized models obtained by our approach have a maximum latency of 29 ms. On average, the inference latency of the models optimized by Avatar is 44× ($\approx (33 + 54)/2$) faster than that of the ones produced by Compressor, which highlights the effectiveness of Avatar in enhancing the usability of language models compared to the state-of-the-art approach.

Avatar also improves the energy consumption of the optimized models by 3× and 8× compared to Compressor on vulnerability prediction and clone detection, respectively. These reductions also translate into a corresponding decrease in carbon footprint, with reductions of 4× and 7× on the two tasks. Overall, except for model size, the models optimized by Avatar outperform the ones optimized by Compressor across all metrics.

Answers to RQ2: Avatar significantly outperforms Compressor (i.e., the state-of-the-art approach) in terms of prediction accuracy (0.75% on average), inference latency (44× faster on average), energy consumption (up to 8× less), and carbon footprint (up to 7× less).

5 DISCUSSIONS

5.1 Efficiency of Avatar

We investigate the time taken by Avatar to optimize language models of code, breaking it down into four parts: pruning the configuration space, building the effectiveness indicator, executing the configuration tuning algorithm, and training optimized models.

In our experimental setup, the parallel execution of pruning the configuration space takes just 5 minutes to complete. After that, Avatar uses a single 16 GB Tesla V100 GPU to train 20 models for constructing the effectiveness indicator, consuming approximately 10 hours. Note that this overhead is only rarely incurred, e.g., the first time optimizing a language model for deployment, which may occur only on a monthly or yearly basis. Because of the carefully pruned configuration space and the specialized optimization algorithm, Avatar efficiently returned Pareto-optimal configurations in about 2 minutes. Subsequently, the knowledge distillation phase required more time, with Avatar taking an average of 14.9 and 18.3 minutes to train an optimized model for the vulnerability prediction and clone detection tasks, respectively. These results underscore the fact that Avatar can produce well-performing optimized models with much less time cost than fine-tuning or pre-training large language models, which often takes a few hours or days [65].

Table 4: Usefulness of Avatar in cloud deployment. The results show how many queries that the models can process per second when deployed on a cloud server.

Model	Vulnerability Prediction	Clone Detection
CodeBERT	58	64
CodeBERT-Avatar	**171 (2.9×)**	**476 (7.4×)**
GraphCodeBERT	79	48
GraphCodeBERT-Avatar	**390 (4.9×)**	**570 (11.9×)**
Average Improvements	**3.9×**	**9.7×**

5.2 Usefulness in Cloud Deployment

The primary goal of Avatar is to optimize language models of code for deployment on developers' personal devices like laptops. As mentioned in Section 1, we hold this perspective due to privacy concerns [36, 48, 57, 80] and the need for use under poor network conditions. Deploying models on cloud servers may not be a viable option because it requires sending code to third-party vendors, which is prohibited by some companies that consider code bases to be important intelligent properties. Also, cloud deployment may result in more inference latency for developers in some regions with poor bandwidth or Internet coverage. However, we acknowledge that cloud deployment is a common practice today, offering more computing resources and scalability to support a larger user base. Therefore, it would be worthwhile to also discuss the benefits of optimized models in the context of cloud deployments.

We run experiments assuming that the models process queries in batch mode with a batch size of 100. These experiments are run on a server equipped with a Tesla V100 GPU. We send the queries directly from the GPU's host machine to eliminate any potential impact from network fluctuations, and then measure how many queries the models can process per second. The experimental results, presented in Table 4, show that compared to the original language models of code, the optimized models can process on average 3.9× and 9.7× more queries per second on the two tasks, respectively. These results highlight the advantages of using Avatar for deploying large language models of code in cloud servers.

5.3 Threats to Validity

One potential threat to internal validity is the randomness inherent in the configuration tuning algorithms used in our experiments. To address this concern, we have run each experiment 10 times and reported the average results, as recommended by Arcuri and Briand [3]. Regarding external validity, a potential threat is that our results may not be generalizable to other models and tasks beyond the ones we have studied. To ensure the generalizability of our work, we have carefully selected two representative encoder-only language models of code and two popular downstream tasks with different characteristics for our evaluation. This ensures that our results are unbiased and our method potentially applies to a broad context. While we have not yet applied our method to other types of language models, such as decoder-only models, which have also recently gained popularity, we plan to extend our study on those models to further validate our work's generalizability in the future. One threat to construct validity is that the evaluation metrics may not fully capture the performance of our Avatar and the baseline in enhancing the usability and sustainability of language models of code. To mitigate it, we use a total of five widely-used evaluation metrics to compare the effectiveness of Avatar and the baseline from a comprehensive set of perspectives.

6 RELATED WORK

In recent years, both the natural language processing and software engineering communities have dedicated their efforts to optimizing language models. However, unlike our work, which seeks to simultaneously optimize multiple aspects of language models of code, most existing studies focus on reducing model size only, thereby indirectly mitigating other related issues such as inference latency. These existing studies typically fall into three main categories: model pruning, model quantization, and knowledge distillation.

Model pruning and quantization involve directly altering model parameters to reduce model size. Model pruning replaces certain parameters with zeros, or removes network components like hidden layers [16, 54]. Model quantization converts a model's 32-bit floating-point parameters into lower-bit fixed-point values [19, 43, 81]. These techniques have proven effective in reducing model size to a level suitable for deployment in scenarios with less stringent requirements. A recent study has also demonstrated their potential to reduce the computational cost and carbon footprint of language models of code [75], offering a promising avenue for future research. However, these techniques fall short of meeting the 3 MB model size recommendation put forth by Svyatkovskiy et al. [70] within the context of software engineering. As a result, we have chosen not to include them in our pipeline and comparison experiments.

We have introduced knowledge distillation in Section 2, an essential step in Avatar and the baseline. While several knowledge distillation methods have been proposed, most of them typically result in models ranging from 100 to 200 MB [41, 60, 68, 77]. Some studies [8, 71, 78, 84] have successfully optimized language models into sizes ranging from 20 to 40 MB. Notably, only Compressor [65] has achieved the remarkable feat of optimizing a large language model of around 500 MB into a compact 3 MB model. Therefore, we only compare Avatar with Compressor in our experiments.

The software engineering research community has also explored alternative methods for optimizing language models of code. For example, Grishina et al. [25] propose using only the initial layers of language models during inference to reduce resource consumption. Additionally, Zhang et al. [83] introduce a technique to simplify the input programs for CodeBERT, significantly reducing computational cost without compromising model performance. Despite these efforts, there are still gaps in optimizing language models of code to simultaneously improve usability and environmental sustainability. To the best of our knowledge, our study is the first to address both aspects concurrently.

7 CONCLUSION AND FUTURE WORK

This paper proposes Avatar, a novel approach that can optimize large language models of code in terms of model size, inference latency, energy consumption, and carbon footprint without sacrificing effectiveness (e.g., prediction accuracy on downstream tasks) by much, thereby improving the usability and environmental sustainability of language models of code. The key idea of Avatar is to formulate the optimization of language models as a multi-objective configuration tuning problem and solve it with the help of SMT solvers and a tailored optimization algorithm. We evaluate Avatar with two state-of-the-art language models, i.e., CodeBERT and GraphCodeBERT, on two popular tasks, i.e., vulnerability prediction and clone detection. We use Avatar to produce optimized models with a small size (3 MB), which is 160× smaller than the original large models. On the two tasks, the optimized models can significantly reduce the energy consumption (up to 184× less), carbon footprint (up to 157× less), and inference latency (up to 76× faster), with only a negligible loss in effectiveness (1.67% on average). Compared with the state-of-the-art approach, Avatar optimizes language models of code more effectively in all metrics.

In the future, we plan to further investigate the effectiveness and efficiency of our proposed approach Avatar by experimenting with more large language models of code beyond those considered in this paper, such as the generative language models of code.

Replication Package: The code, datasets, and documentation for this work, along with all obtained models, are available via this link: https://github.com/soarsmu/Avatar.

ACKNOWLEDGMENTS

This research / project is supported by the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 143–154.

[2] Akvelon. 2023. Code Search: a Closer Look at Akvelon's Source Code Search Engine — akvelon.com. https://akvelon.com/code-search-a-closer-look-at-akvelons-source-code-search-engine/. [Accessed 28-09-2023].

[3] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*. 1–10.

[4] Gareth Ari Aye and Gail E Kaiser. 2020. Sequence model design for code completion in the modern IDE. *arXiv preprint arXiv:2004.05249* (2020).

[5] Lei Jimmy Ba and Rich Caruana. 2014. Do deep nets really need to be deep?. In *Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 2*. 2654–2662.

[6] Nikolaj Bjørner. 2013. SMT in verification, modeling, and testing at microsoft. In *Hardware and Software: Verification and Testing: 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers 8*. Springer, 3–3.

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[8] Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. 2020. AdaBERT: Task-Adaptive BERT Compression with Differentiable Neural Architecture Search. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. 2463–2469.

[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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] Tao Chen and Miqing Li. 2023. The weights can be harmful: Pareto search versus weighted search in multi-objective search-based software engineering. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–40.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[12] GitLab Auto DevOps. 2023. Top 10 ways machine learning may help DevOps — about.gitlab.com. https://about.gitlab.com/blog/2022/02/14/top-10-ways-machine-learning-may-help-devops/. [Accessed 22-09-2023].

[13] Thomas Dohmke, Marco Iansiti, and Greg Richards. 2023. Sea Change in Software Development: Economic and Productivity Analysis of the AI-Powered Developer Lifecycle. *arXiv preprint arXiv:2306.15033* (2023).

[14] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2018. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks* 107 (2018), 3–11.

[15] Hugging Face. 2023. Configurations of Encoder-only Models — huggingface.co. https://huggingface.co/docs/transformers/model_doc/roberta#transformers.RobertaConfig. [Accessed 25-09-2023].

[16] Angela Fan, Edouard Grave, and Armand Joulin. 2020. Reducing Transformer Depth on Demand with Structured Dropout. In *2020 8th International Conference on Learning Representations*.

[17] Yongsheng Fang and Jun Li. 2010. A review of tournament selection in genetic programming. In *International symposium on intelligence computation and applications*. Springer, 181–192.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547.

[19] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Hassan Sajjad, Preslav Nakov, Deming Chen, and Marianne Winslett. 2021. Compressing Large-Scale Transformer-Based Models: A Case Study on BERT. *Transactions of the Association for Computational Linguistics* 9 (09 2021), 1061–1080.

[20] Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2023. Runtime performance prediction for deep learning models with graph neural network. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 368–380.

[21] Yanjie Gao, Yonghao Zhu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2021. Resource-guided configuration space reduction for deep learning models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 175–187.

[22] GitHub. 2023. GitHub Copilot Community. https://github.com/orgs/community/discussions/categories/copilot?discussions_q=category%3ACopilot+network. [Accessed 03-10-2023].

[23] GitHub. 2023. GitHub Copilot · Your AI pair programmer — github.com. https://github.com/features/copilot/. [Accessed 22-09-2023].

[24] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129, 6 (2021), 1789–1819.

[25] Anastasiia Grishina, Max Hort, and Leon Moonen. 2023. The EarlyBIRD Catches the Bug: On Exploiting Early Layers of Encoder Models for More Efficient Code Classification. *arXiv preprint arXiv:2305.04940* (2023).

[26] Daya Guo, Shuo Ren, 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, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *2021 9th International Conference on Learning Representations*.

[27] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-efficient performance learning for configurable systems. *Empirical Software Engineering* 23 (2018), 1826–1867.

[28] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance prediction for configurable software with deep sparse neural network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1095–1106.

[29] Mirazul Haque, Yaswanth Yadlapalli, Wei Yang, and Cong Liu. 2022. EREBA: Black-Box Energy Testing of Adaptive Neural Networks. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 835–846.

[30] Kazuyuki Hara, Daisuke Saito, and Hayaru Shouno. 2015. Analysis of function of rectified linear unit used in deep learning. In *2015 international joint conference on neural networks (IJCNN)*. IEEE, 1–8.

[31] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. 2019. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 960–970.

[32] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* (2016).

[33] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 596–607.

[34] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. In *2015 NIPS Deep Learning and Representation Learning Workshop*.

[35] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv preprint arXiv:2308.10620* (2023).

[36] Yizhan Huang, Yichen Li, Weibin Wu, Jianping Zhang, and Michael R Lyu. 2023. Do Not Give Away My Secrets: Uncovering the Privacy Issue of Neural Code Completion Tools. *arXiv preprint arXiv:2309.07639* (2023).

[37] Zhiheng Huang, Davis Liang, Peng Xu, and Bing Xiang. 2020. Improve Transformer Models with Better Relative Position Embeddings. *Findings of the Association for Computational Linguistics: EMNLP 2020* (2020).

[38] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[39] Yasir Hussain, Zhiqiu Huang, Yu Zhou, Izhar Ahmed Khan, Nasrullah Khan, and Muhammad Zahid Abbas. 2023. Optimized Tokenization Process for Open-Vocabulary Code Completion: An Empirical Study. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 398–405.

[40] Apple Inc. 2023. MacBook Pro 14- and 16-inch - Tech Specs — apple.com. https://www.apple.com/sg/macbook-pro-14-and-16/specs/. [Accessed 03-10-2023].

[41] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 4163–4174.

[42] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1073–1085.

[43] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. I-BERT: Integer-only BERT Quantization. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 5506–5518.

[44] Padmavathi Kora and Priyanka Yadlapalli. 2017. Crossover operators in genetic algorithms: A review. *International Journal of Computer Applications* 162, 10 (2017).

[45] Taku Kudo. 2018. Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics.

[46] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. 2019. Quantifying the Carbon Emissions of Machine Learning. *arXiv preprint arXiv:1910.09700* (2019).

[47] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.

[48] David Lo. 2023. Trustworthy and Synergistic Artificial Intelligence for Software Engineering: Vision and Roadmaps. *arXiv preprint arXiv:2309.04142* (2023).

[49] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[50] Sean Luke. 2009. *Essentials of metaheuristics*.

[51] David JC MacKay. 1992. Bayesian interpolation. *Neural Computation* 4, 3 (1992), 415–447.

[52] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. 2016. An Empirical Study of Practitioners' Perspectives on Green Software Engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 237–248.

[53] Ivan Mehta. 2023. Apple reportedly limits internal use of AI-powered tools | TechCrunch. https://techcrunch.com/2023/05/19/apple-reportedly-limits-internal-use-of-ai-powered-tools-like-chatgpt-and-github-copilot. [Accessed 03-10-2023].

[54] Paul Michel, Omer Levy, and Graham Neubig. 2019. Are Sixteen Heads Really Better than One?. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[55] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, Luc De Raedt (Ed.). ijcai.org, 5546–5555.

[56] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2136–2148.

[57] Liang Niu, Shujaat Mirza, Zayd Maradni, and Christina Pöpper. 2023. {CodexLeaks}: Privacy Leaks from Code Generation Language Models in {GitHub} Copilot. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2133–2150.

[58] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[59] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[60] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[61] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. 2020. Green ai. *Commun. ACM* 63, 12 (2020), 54–63.

[62] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *54th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics (ACL), 1715–1725.

[63] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics.

[64] Jieke Shi, Zhou Yang, Junda He, Bowen Xu, and David Lo. 2022. Can identifier splitting improve open-vocabulary language model of code?. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1134–1138.

[65] Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang, and David Lo. 2022. Compressing pre-trained models of code into 3 mb. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[66] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C Briand, and Frank Zimmer. 2018. Test case prioritization for acceptance testing of cyber physical systems: a multi-objective search-based approach. In *Proceedings of the 27th acm sigsoft international symposium on software testing and analysis*. 49–60.

[67] Jeffrey M Stanton. 2001. Galton, Pearson, and the peas: A brief history of linear regression for statistics instructors. *Journal of Statistics Education* 9, 3 (2001).

[68] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019. Patient Knowledge Distillation for BERT Model Compression. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 4323–4332.

[69] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480.

[70] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and Memory-Efficient Neural Code Completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 329–340.

[71] Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. 2019. Distilling task-specific knowledge from bert into simple neural networks. *arXiv preprint arXiv:1903.12136* (2019).

[72] Michael E Tipping. 2001. Sparse Bayesian learning and the relevance vector machine. *Journal of machine learning research* 1, Jun (2001), 211–244.

[73] Takahisa Toda and Takehide Soh. 2016. Implementing efficient all solutions SAT solvers. *Journal of Experimental Algorithmics (JEA)* 21 (2016), 1–44.

[74] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[75] Xiaokai Wei, Sujan Gonugondla, Shiqi Wang, Wasi Ahmad, Baishakhi Ray, Haifeng Qian, Xiaopeng LI, Varun Kumar, Zijian Wang, Yuchen Tian, Qing Sun, Ben Athiwaratkun, Mingyue Shang, Murali Krishna Ramanathan, Parminder Bhatia, and Bing Xiang. 2023. Towards greener yet powerful code generation via quantization: An empirical study. In *ESEC/FSE 2023*.

[76] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[77] Canwen Xu, Wangchunshu Zhou, Tao Ge, Furu Wei, and Ming Zhou. 2020. BERT-of-Theseus: Compressing BERT by Progressive Module Replacing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 7859–7869.

[78] Jin Xu, Xu Tan, Renqian Luo, Kaitao Song, Jian Li, Tao Qin, and Tie-Yan Liu. 2021. NAS-BERT: Task-Agnostic and Adaptive-Size BERT Compression with Neural Architecture Search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Virtual Event, Singapore). Association for Computing Machinery, New York, NY, USA, 1933–1943.

[79] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-Trained Models of Code. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1482–1493.

[80] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2023. What Do Code Models Memorize? An Empirical Study on Large Language Models of Code. *arXiv preprint arXiv:2308.09932* (2023).

[81] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 811–824.

[82] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An Extensive Study on Pre-Trained Models for Program Understanding and Generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 39–51.

[83] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1073–1084.

[84] Sanqiang Zhao, Raghav Gupta, Yang Song, and Denny Zhou. 2021. Extremely Small BERT Models from Mixed-Vocabulary Training. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. Association for Computational Linguistics, Online, 2753–2759.

[85] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing generalizability of codebert. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 425–436.

[86] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc.