Python Code Generation using Pre-trained Models

Md Robiul Islam
Department of Computer Science
William & Mary
miislam@wm.edu

***Abstract*—Recent advancements in natural language processing (NLP) have demonstrated near-human performance across a wide range of tasks, opening new possibilities for their application in structured domains with strict syntax rules, such as programming languages. This paper explores the potential of pre-trained language models for Python code generation, leveraging their ability to understand and generate syntactically and semantically correct code. We explored some pre-trained models for Python code generation. Our approach achieves a BLEU score of 30.78 for the fine-tuned CodeLlama-7b model, demonstrating the model's capability to generate accurate and meaningful Python code from natural language descriptions. Additionally, we provide a comprehensive pipeline for data preprocessing and model training, ensuring reproducibility and accessibility. All code, datasets, and results are publicly available to foster further research and innovation in NLP and code generation. The code is available here: Github.**

***Index Terms*—Code generation, Python, BLEU, Fine-tuned**

I. Introduction

The most intriguing area of artificial intelligence, deep learning [1], is motivated by biological neural networks to identify patterns in the actual world. Large amounts of data may be processed and learned from by these neural networks with amazing efficiency. Image, video, text, and voice analysis all make use of it. The pre-trained language model is a neural network that has been trained to transform a string of input words into a context-dependent embedding using unlabeled text data, such as in an unsupervised, task-agnostic manner. Conversely, the idea of fine-tuning the pre-trained model involves training a pre-trained model that has already been initialized utilizing the weights of this pre-trained language model as the starting weights for the subsequent training.

The first pre-trained language models were proposed to be built on top of recurrent neural networks (RNN) [2]. Additionally, it has been demonstrated that adjusting an RNN-based model on a particular task after pre-training it on unlabeled data yields superior outcomes to directly training a randomly initialized model on the same task. Sequence models based on RNNs have two drawbacks. First, RNNs are unable to fully recall non-sequential tokens since the tokens are processed sequentially, requiring token after token to be processed. Second, long-term relationships between code tokens might not be captured by RNN-based models.

Transformer-based pre-trained language models with various topologies have been introduced by several prominent businesses, including HuggingFace, Microsoft, Google, and OpenAI. In general, transformer models can be divided into three groups. BERT, RoBERTa, ELECTRA, and Luke are examples of encoder-only transformers, which fall within the first type. Decoder-only transformers, including GPT-2, Llama, Falcon, and ChatGPT, are examples of the second type. MarianMT, T5, and BART encoder-decoder transformer models are included in the third group. In encoder–decoder models, an intermediate representation is created by the encoder processing input sequences via embedding and attention methods. This representation is then used by the decoder to produce an output sequence. Interestingly, models such as BERT and GPT have shown remarkable performance with little fine-tuning, attaining cutting-edge outcomes on a variety of natural language understanding (NLU) tasks.

Several methods are used in the active research area to automate even minor aspects of software development [3]. Numerous software programmers can save time by successfully automating even small activities, which results in resource savings for a variety of sectors. Furthermore, automatic code generation will become more and more crucial as software continues to consume the world and the need for skilled software engineers continues to exceed supply.

In this paper, we propose a machine learning model to automate the task of writing code by assisting developers in writing individual units of functionality (or 'functions'). Automating code generation can take many forms. auto-completing lines of source code to generate lines of source code from comments. In this research, we aim to take the initial lines of code (a function signature) along with a docstring (function documentation) and generate the corresponding function body. To do this, we use a pre-trained language model and fine-tune it on a canonical corpus of Python code scraped from Huggingface [4]. The major contributions of this paper are as follows:

- Fine-tuned three different pre-trained models.
- When the pre-trained models are fine-tuned with docstring as input and code as output and docstring + code as input and code as output. The 2nd approach outperforms the 1st one in terms of BLEU score.
- CodeLlama-7b outperforms the other two pre-trained models such as CodeT5-Large and GPT-2 in terms of Code Generation.

II. Background

The fact that code generation is still an active field of study with numerous potential answers and ongoing research is one of its main challenges [5]. Cutting-edge solutions do not, but

do fall far short of automating routine operations that software workers do every day. With the advent of transformer [6] and the development of pre-training techniques [7], more and more pre-training models are applied in the field of code generation. For instance, open-source code models like CodeT5 [8], CodeT5+ [9], CodeGen [10], PolyCoder [11], InCoder [12], StarCoder [13], as well as general-purpose language models such as GPT-J [14], GPT-Neo [15] have demonstrated substantial performance in code generation tasks.

The dominant approaches in code generation mainly involve fine-tuning pre-trained code generation models using supervised learning [16] or reinforcement learning (RL) [17]. However, neither supervised nor reinforcement learning fine-tuning allows the model to learn reasoning well. Moreover, RL-based approaches decompose code generation into sequences of token-generating actions, which may limit the model to learn reasoning ability due to the lack of high-level thinking. Different from these methods, we achieve high-level thinking in smaller models by distilling the reasoning abilities of LLMs into them.

A. Traditional Code Generation

Token completion based on structured data gleaned from static code analysis is the most conventional and well-known method utilized by numerous IDEs in a variety of languages. For instance, the system will try to identify strings that are nearly identical to function definitions when a developer inputs a string of characters and suggests finishing these function calls. Likewise, for object methods, the IDE will suggest auto-completing various object methods when the accessor token (such as "-" or "."), is typed. The main flaw in these methods is that they do not fully comprehend the programmers' intentions and don't provide context for the surrounding code beyond what the tool's authors' heuristics provide.

B. Using machine learning for Code Generation

Other, more innovative methods from the literature [5] are usually used in limited linguistic domains and have extremely complex evaluation outcomes, among other issues. In particular, programming languages frequently use non-natural variable names, function names, and syntax with more structure, whereas pre-trained models are trained on free-form language data [5]. The goal of this field's work has been to develop more structured models that utilize particular architectures [18]. The developers of [19] begin by breaking down the context's text token input sequence into a structure resembling a tree. Other strategies include limiting the model's output to a domain-specific language (DSL) or context-free grammar (CFG) [20]. For the output of a code generation model to be syntactically correct, it must follow a fairly precise form.

Instead, in this paper, we concentrate on adopting an alternative strategy. We focus on enhancing performance on the code prediction job by utilizing pre-trained language models that are then refined on code, as seen by the steadily growing sizes of language models. Figure 1 demonstrates the code generation from docstring.

C. Using Pre-train model for code generation

The capacity of pre-trained models to generalize across different codebases and programming languages is their main benefit in code generation [21]. Pre-trained models already have a large body of information, so they don't have to start from scratch and can better understand and produce code in a variety of programming paradigms. Since these models can move between languages with ease and don't require a lot of language-specific training, this generalization is essential for handling multilingual code production tasks. The decrease in the amount of time and resources needed for training is another important advantage. With comparatively little domain-specific data, researchers and developers can apply the expertise gained from fine-tuning pre-trained models on large corpora to particular code production tasks. Compared to training from scratch, fine-tuning an existing model uses less data and processing power, making it more widely available. Additionally, when creating code, pre-trained models demonstrated their ability to comprehend the surrounding context and capture contextual dependencies. Context awareness aids in the creation of more logical and semantically significant code fragments. Pre-trained models may efficiently handle complicated syntactic structures found in code generation jobs by identifying complex patterns and dependencies in code expressions.

D. Dataset and Feature

We are using the CodeSearchNet dataset in this project [4]. Two million (comment, code) pairs from open-source libraries in languages including Python, Javascript, PHP, Java, Go, and Ruby make up the dataset. With a 95% codelength of up to 350 tokens, the median codelength is 60–100 text tokens. Ten text tokens make up the median length of the documentation. Instead of including samples from other programming languages, we limit our collection to Python samples. With a focus on Python, our dataset consists of over 500k (comment, code) pairs and more than 1M methods. This choice is based on modeling as well as practical considerations. Figure 2 shows the data distribution of the CodeSearchNet dataset.

Practically speaking, more in-depth ablation investigations are made possible by limiting to a dataset of a manageable size that is concentrated on a single language domain. From the standpoint of modeling, we think it is simpler to transfer learning from a natural language to a computer language like Python.

III. Methodology

In this section, we explain the overall methodology of our work.

A. CodeLlama

Code Llama is a collection of pre-trained and fine-tuned generative text models ranging in scale from 7 billion to 34 billion parameters. Code Llama comes in three model sizes, and three variants.

Code Llama: base models designed for general code synthesis

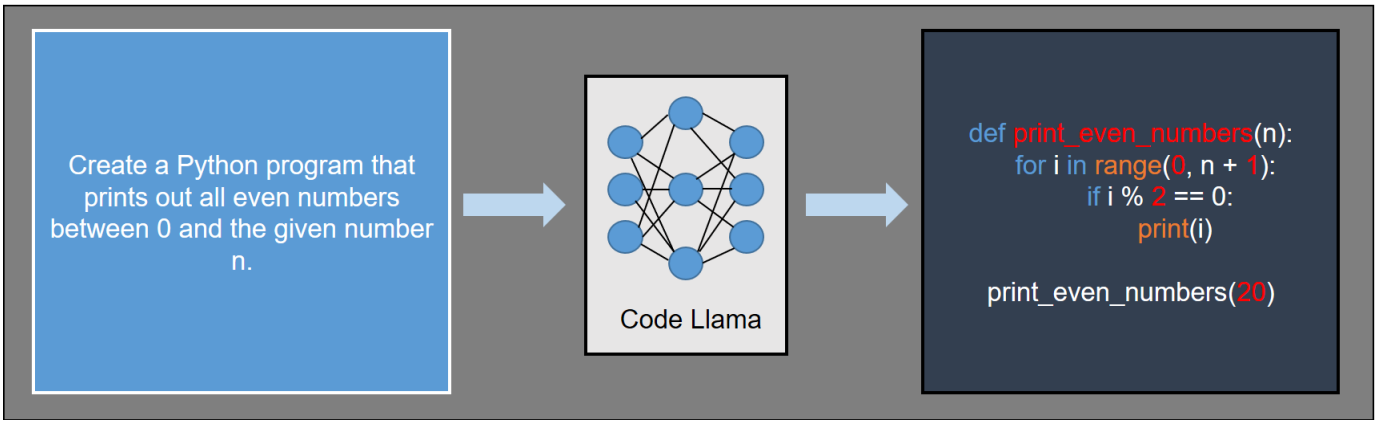

Fig. 1. Example of generating code from docstring

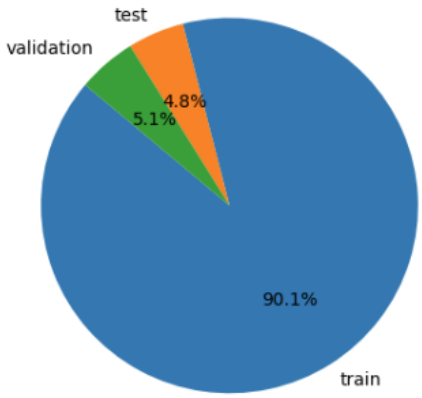

Fig. 2. Python data distribution

and understanding.

Code Llama - Python: designed specifically for Python.

Code Llama - Instruct: for instruction following and safer deployment.

All variants are available in sizes of 7B, 13B, and 34B parameters.

B. CodeT-5

CodeT5-Large is a transformer-based pre-trained model from Salesforce designed for a wide variety of code-related tasks, including code generation, summarization, translation, and classification. It is part of the CodeT5 family, which is fine-tuned specifically on programming languages like Python, Java, and others. CodeT5-Large is a larger variant with more parameters, which often leads to better performance on complex tasks compared to smaller versions. CodeT5 is a family of encoder-decoder language models for code from the paper [22]. There is a 770M parameter in this model.

C. GPT-2

GPT-2 [23] is a transformers model pre-trained on a very large corpus of English data in a self-supervised fashion. This means it was pre-trained on the raw texts only, with no humans labeling them in any way (which is why it can use lots of publicly available data) with an automatic process to generate inputs and labels from those texts. More precisely, it was trained to guess the next word in sentences.

More precisely, inputs are sequences of continuous text of a certain length and the targets are the same sequence, shifting one token (word or piece of word) to the right. The model uses internally a masking mechanism to make sure the predictions for the token.

This way, the model learns an inner representation of the English language that can then be used to extract features useful for downstream tasks. The model is best at what it was trained for, however, which is generating texts from a prompt. This is the smallest version of GPT-2, with 124M parameters.

D. Fine-Tuned Pre-Trained Large Language Models

To fine-tune the model, we need to optimize the model. Because, models like CodeLlama which has almost 7B parameters and if we want to finetune this, then we need to update all the weight of the model which is 7B. This is a very time-consuming as well as not inefficient approach in terms of GPU. Here comes LORA [24], LORA is a parameter-efficient technique to optimize the model for fine-tuning. LoRA reduces the number of trainable parameters by learning pairs of rank-decomposition matrices while freezing the original weights. This vastly reduces the storage requirement for large language models adapted to specific tasks and enables efficient task-switching during deployment all without introducing inference latency. LoRA also outperforms several other adaptation methods including adapter, prefix-tuning, and fine-tuning.

IV. Result

We fine-tune 3 different models for code generation. We have chosen three evaluation metrics to evaluate our model: BLUE, ROUGE, and BERT. CodeLlama (7B) outperforms the other two models based on BLEU, ROUGE, and BERT scores. The scores are 30.78, 0.38 and 0.87 respectively. There could be several reasons why CodeLlama outperforms other models such as the CodeLlama is a decoder-based model and that's why the model is not so complex. Also, the

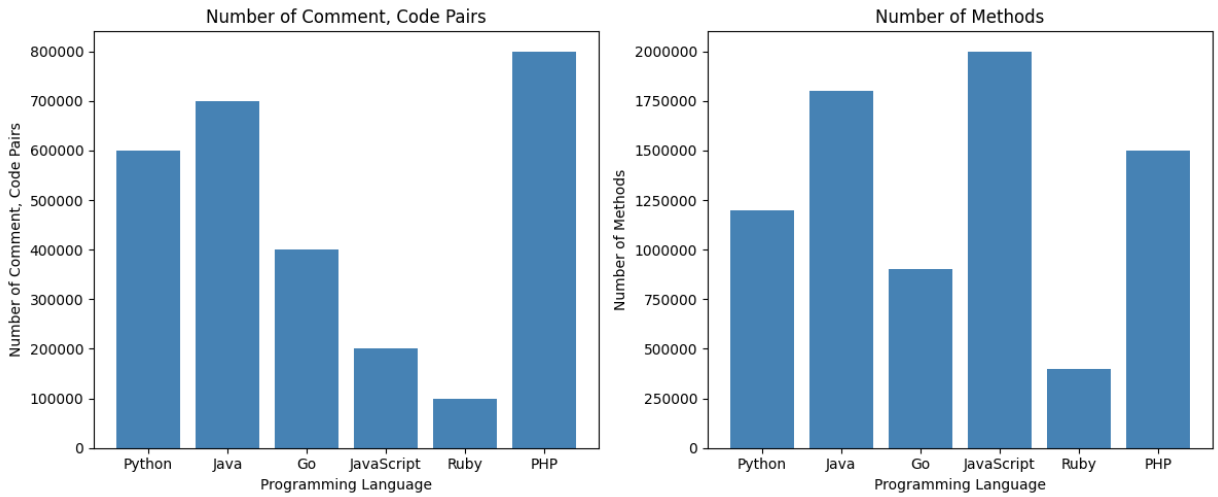

Fig. 3. Histogram of the number of (comment, code) pairs available in our dataset, as well as the number of unique function methods for each language

TABLE I
PARAMETER FOR FINE-TUNE THE MODEL

Parameter name	Value
num_train_epochs	3
evaluation_strategy	”epoch”
learning_rate	2e-5
logging_steps	1e-10
per_device_train_batch_size	16
per_device_eval_batch_size	16
save_total_limit	3
fp16	True
weight_decay	0.01
r	16
lora_alpha	64
lora_dropout	0.01

context length of CodeLlama is far higher than the other two models. Furthermore, CodeLlama is pre-trained with different programming languages. So, the model already knows about the Python code. In the Table II, all the score is presented.

TABLE II
EVALUATION

Pre-train model	BLUE score	ROUGE-L	BERT score
CodeLlama (7B)	**30.78**	0.38	0.87
CodeT-5 (Large)	14.56	0.31	0.59
GPT-2	18.83	0.28	0.85

```
def is_prime(number):
    """
    Check if a number is prime.

    Parameters:
    - number: int
     The number to check.

    Returns:
    - bool
     True if the number is prime, False otherwise.
    """
    if number <= 1:
        return False
    if number <= 3:
        return True
    if number % 2 == 0 or number % 3 == 0:
        return False

    i = 5
    while i * i <= number:
        if number % i == 0 or number % (i + 2) == 0:
            return False
        i += 6

    return True
```

Listing 1. Fine-tune input is code + docstring

We also observed that when we fine-tune the model with only docstring which is present in list 2 as input and code is output the model performs well. However, when we fine-tune the model with an input like code + docstring which is present in list 1 then the model outperforms the only input which is docstring.

```
    """Check if a number is prime."""
```

Listing 2. Fine-tune input is only docstring

V. CONCLUSION

In this paper, we explore automating code generation from docstring or human text. We found the best-performing model is CodeLlama which has been trained with a large Python dataset. Even though our model only focuses on generating Python code. We can achieve a BLEU score of 30.78 which outperforms [25].

VI. LIMITATION

Even though CodeLlama achive a higher BLEU score but still there is a limitations of our work. We only focus on only one programming language, we accept that CodeLlama

outperform for generating python code from docstring but it will not perform very well to generate other programming languages such as Java. In our future work, we plan to explore other programming languages with other models.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] A. M. Dai and Q. V. Le, "Semi-supervised sequence learning," *Advances in neural information processing systems*, vol. 28, 2015.

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[4] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[5] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Codegru: Context-aware deep learning with gated recurrent unit for source code modeling," *Information and Software Technology*, vol. 125, p. 106309, 2020.

[6] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[7] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[8] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[9] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[10] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[11] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[12] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[13] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[14] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model," https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[15] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang *et al.*, "Gpt-neox-20b: An open-source autoregressive language model," *arXiv preprint arXiv:2204.06745*, 2022.

[16] D. Hendrycks, K. Zhao, S. Basart, J. Steinhardt, and D. Song, "Natural adversarial examples," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 15 262–15 271.

[17] K. Tan, W. Fan, and Y. Wei, "Hybrid reinforcement learning breaks sample size barriers in linear mdps," *arXiv preprint arXiv:2408.04526*, 2024.

[18] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 05, 2020, pp. 8984–8991.

[19] X. Liu, X. Kong, L. Liu, and K. Chiang, "Treegan: syntax-aware sequence generation with generative adversarial networks," in *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2018, pp. 1140–1145.

[20] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7055–7062.

[21] A. Soliman, S. Shaheen, and M. Hadhoud, "Leveraging pre-trained language models for code generation," *Complex Intelligent Systems*, 2024.

[22] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[23] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[24] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=nZeVKeeFYf9

[25] L. Perez, L. Ottens, and S. Viswanathan, "Automatic code generation using pre-trained language models," *arXiv preprint arXiv:2102.10535*, 2021.