Evaluating Large Language Models on Code Generation

Daniel da Costa
danielp3@usc.edu

Karen Wang
kwang760@usc.edu

Qasim Riaz Siddiqui
qsiddiqu@usc.edu

Saurav Joshi
syjoshi@usc.edu

Abstract

In this study, the performances in Python code generation of three different code generation models – CodeT5, CodeGen, and GPT-3.5 – were compared using the Mostly Basic Python Problems (MBPP) dataset. The *pass@k* metric was used as the primary method of evaluation, and CodeT5 and CodeGen were evaluated in a few-shot setting, while GPT-3.5 was evaluated in zero-shot and few-shot settings. The findings suggest that GPT-3.5 performs best in the few-shot setting, followed by GPT-3.5 in the zero-shot setting, CodeT5 in the few-shot setting, and CodeGen in the few-shot setting. These results indicate that GPT-3.5 is a promising model for code generation tasks, particularly in situations where the training data is limited, and highlight the importance of providing contextual information through prompting to overcome certain deficiencies in the zero-shot setting. GitHub repo: https://github.com/sauravjoshi23/CodeGPT

1 Introduction

Code generation has been the subject of extensive research in recent years, and various models have been developed to generate code snippets for different programming tasks. However, the effectiveness of these models often depends on the specific task performed and the available training data. Thus, there is a need to evaluate the effectiveness of different code generation models on limited training data and identify the most promising models for different tasks. This study aims to address this need by comparing the performances of CodeT5, CodeGen, and GPT-3.5 on natural language text examples from the MBPP dataset (Austin et al., 2021) to provide insights into the effectiveness of these models in Python code generation and help inform the development of more efficient and accurate code generation models.

2 Related work

Recent research has focused on neural language models, which have shown promise in generating code snippets for different programming tasks. Notable examples of neural language models are GPT-3 and GPT-3.5, which have demonstrated impressive performance on a range of natural language tasks, including code generation (Brown et al., 2020). GPT-3 and GPT-3.5 use a transformer architecture and are pre-trained on a large corpus of text to improve the quality of generated code snippets. Recent work has also explored variations of the transformer architecture that incorporate domain-specific knowledge. These architectures are employed by models such as CodeT5 (Wang et al., 2021), and CodeGen (Nijkamp et al., 2023). CodeT5 is a code generation model that is based on the T5 transformer architecture and was specifically designed for code generation tasks. It was trained on a large-scale dataset called CodeSearchNet (Husain et al., 2019), which includes over 6 million lines of code from six programming languages and various libraries. CodeGen is a code generation model that uses a combination of program synthesis and neural language modeling techniques to generate code snippets for a variety of programming tasks. It was trained on THEPILE, BIGQUERY, and BIGPYTHON datasets. The CodeGen family of models contains 2B, 6B, and 16B parameter models, but due to computational constraints, the 2B model was leveraged (the 6B model required more than 16B of GPU).

3 Problem Description

Large language models can capture the syntax, semantics, and structure of programming

languages and generate code snippets that are close to human-written code. However, the effectiveness of large language models for code generation may vary depending on the specific task and the available training data. Some code generation tasks require a significant amount of training data and fine-tuning to achieve optimal performance, while others may require the use of specialized models or techniques. Moreover, large language models are complex and require significant computational resources to train and optimize. The size and complexity of these models can make them challenging to use in real-world scenarios, where computational resources may be limited. This study compares the performances of several code generation models to reveal their strengths and weaknesses and discover ways in which they can be improved.

4 Methods

The input requirements for the three code generation models varied slightly. The CodeT5 model required the natural language task description from the MBPP dataset along with three assert statements (test cases) that demonstrate the desired functionality of the code.

```
Write a python function to remove first
and last occurrence of a given
character from the string. Your code
should satisfy these tests:
 assert remove_Occ("hello","l") == "heo"
 assert remove_Occ("abcda","a") == "bcd"
 assert remove_Occ("PHP","P") == "H"
```

Figure 1: CodeT5 input format

In contrast, the CodeGen model required only one assert statement (test case) along with the task description from the MBPP dataset. When three examples were provided the performance of model decreases drastically, not being able to generate any code. As a result, we evalute the model by passing only one example.

```
Write a python function to remove first
and last occurrence of a given
character from the string.
>>> Example: remove_Occ("hello","l") == "heo"
```

Figure 2: CodeGen input format

The GPT-3.5 zero-shot model required only the task description from the MBPP dataset during inference, while the GPT-3.5 few-shot model required the task description along with three assert statements (test cases). These differences in input requirements stem from the different processing methods used by the models. The CodeT5 and CodeGen models are based on transformer architectures that have been trained on specific datasets with examples and labels, which are then used during inference to generate the code snippets. The GPT-3.5 models, on the other hand, are based on a large, pre-trained transformer architecture that can generate text output without additional training or examples.

An additional processing step was required for the GPT-3.5 zero-shot setting. When asked to output a Python function that fulfilled the requirements of the docstring input, the model not only produced the code, but also provided details on the logic of the code and examples of how to use the function. Thus, the output was post-processed to extract the required code snippet. Furthermore, most Python functions generated by the model did not have the same names as the test cases provided by the MBPP dataset. The generated functions had to be renamed for tests to be successfully run.

During inference, the resource requirements for different code generation models also varied. The pre-trained GPT-3.5 model was provided by OpenAI and accessed through its API. It was run locally and did not require any GPU resources. To run the CodeGen and CodeT5 models, GPU resources were required. A VM instance with GPU was created on Google Cloud Platform, specifically using an NVIDIA V100. However, even with the high-end NVIDIA V100, the multiple outputs inference for the CodeGen 6B model could not be run. As a result, the CodeGen 2B model was used, which is less computationally intensive but still capable of generating code snippets. These resource limitations highlight the challenges involved in running complex machine learning models for code generation, and the need for careful resource planning and optimization to ensure optimal performance.

5 Experimental Results

The dataset used to evaluate the code generation performances of the three models was the

MBPP dataset. The dataset comprises approximately 1,000 crowd-sourced Python programming problems, where each problem contains a docstring of natural language text detailing the functionality of the code snippet, the code solution, and three assert statements (test cases). All three models were evaluated on the first 100 samples from the MBPP dataset.

The *pass@k* metric was used as the primary method of evaluation, as it is commonly used to evaluate the performance of code generation models (Chen et al., 2021). It robustly estimates the probability that one of k generations passes the tests where n is the generated candidates. For example, if *pass@5* is 80%, it means that for 80% of the queries, the system returned at least one code snippet that passed all test statements among the top 5 results. The performance of CodeT5 and CodeGen was assessed using k values of 1, 2, and 5, while GPT-3.5 was only evaluated with k=1 due to the API's limitations in generating multiple outputs.

	pass@k		
	k=1	k=2	k=5
CodeT5	0.120	0.171	0.229
CodeGen	0.082	0.092	0.100
GPT(zero-shot)	0.40	-	-
GPT(few-shot)	0.69	-	-

Table 1: *Pass@k* metric results

BLEU score, which is also a common metric used to evaluate the quality of machine translation output, was not an appropriate metric for evaluating the performance of code generation models. BLEU score measures the overlap between a machine-generated text and one or more reference texts, based on n-gram matching. However, in the case of code generation, the generated code may differ significantly from the reference code even if it performs the same function, as there can be multiple ways of implementing the same function in code and the generated code may have a different structure or syntax from the reference code.

The outputs of the models were categorized into six different qualitative outcomes: RuntimeError, CompileError, FailedTest, NoResult, PassedOneOrTwoTests, and PassedAllTests (Le et al., 2022). In the RuntimeError outcome, an error occurred in the execution of the generated code, while in the CompileError outcome, an error occurred in program compilation. The NoResult outcome indicates that the model did not output any code at all, which may be due to vagueness in the task description in which the model required more information to be able to provide any meaningful results. The NoResult outcome was kept separate from the FailedTest outcome, in which the model was able to make sense of the task description and output the desired function, but failed the test cases because the outputs from the functions were not the in format that the tests required, or because the model assumed the input(s) to the functions to be in a format different to what the functions in the test cases expected.

For examples of the errors, please see Appendix A.

5.1 GPT-3.5

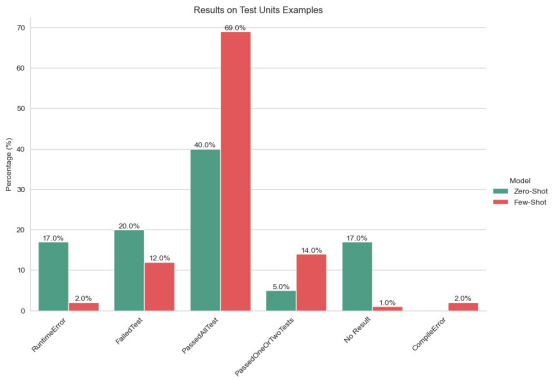

Figure 3: Zero and Few-Shot GPT-3.5: Errors Distribution

We evaluated the gpt-3.5-turbo model on the dataset in both the zero-shot and few-shot settings to understand whether contextual information through prompting helps the model perform better in this scenario.

Figure 3 shows that the model performs really well even in the zero-shot setting, passing all test cases for 40% of the sample. However, we realized that there were some limitations in this setting. These included cases where the model required more information to output any code and cases where the model displayed a logically correct function, but still failed the tests due to differences in output formats. To deal with such cases and more, we evaluated

the model in the few-shot setting, giving it the test cases to learn from.

In this setting, not only did we observe a considerable increase in performance, with the model passing at least one test for 83% of the sample, but we also benefited from a reduction in the post-processing work (alluded to in the "Methods" section) as the model was able to understand the function name, input(s) and output(s) formats from the examples themselves. With the additional prompts, the model was able to understand the context and requirements surrounding the task descriptions as well, which invariably helped to considerably reduce the "No Result" category from 17 to 1. There were, however, certain instances where the model could not display good results even in this setting. This included instances where the task description required the model to understand certain mathematical concepts or where the arrangement of the inputs and outputs of the model did not align perfectly with the test cases (additional error cases are provided in the appendix). We believe that for many of these cases, we can improve the model performance by providing it with even richer details in the prompts and through reinforcement learning with human feedback - ideas to make use of in future work.

5.2 CodeT5

The results of the CodeT5 model evaluated in the few-shot setting are shown in Figure 4.

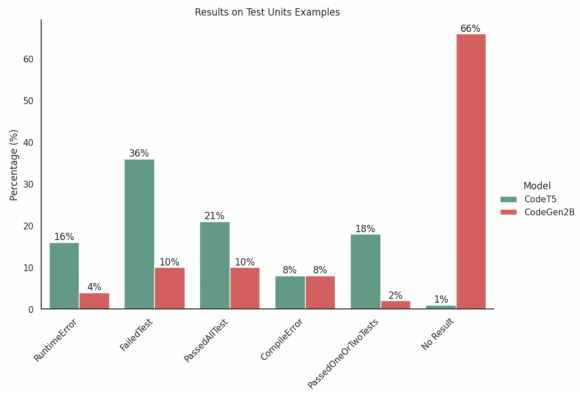

Figure 4: CodeT5 and CodeGen: Errors Distribution

The results indicate that the runtime and compilation errors for code generated by the CodeT5 model in the few-shot setting are relatively high compared to that generated by the GPT-3.5 model in zero-shot and few-shot settings. This implies that CodeT5 is a weaker code generation model than GPT-3.5, as the code it generates may have more syntax errors that the model is unable to notice and address. Additionally, CodeT5 has a higher number of FailedTest outcomes than GPT-3.5, which indicates that although CodeT5 was also given three test cases along with the task description as input, it was unable to leverage the format of the test cases to reduce the formatting issues it produced.

5.3 CodeGen

The results of the CodeGen model with 2B parameters evaluated in the few-shot setting are shown in Figure 4.

Most noticeably, the CodeGen model was unable to generate code for significantly more samples than GPT-3.5 and CodeT5. This is likely due to the fact that CodeGen was trained on specific datasets with examples and labels, which were then used during inference to generate code. The MBPP dataset may be dissimilar to the datasets that it was trained on, thereby the model is likely unable to make sense of the task description objectives to produce any code.

6 Conclusions and Future Work

From this study, GPT-3.5 in the few-shot setting was found to have the best performance for Python code generation, followed by GPT-3.5 in the zero-shot setting, CodeT5 in the few-shot setting, and CodeGen in the few-shot setting. Based on these findings, it can be concluded that providing more context and information to large language models such as GPT-3.5 is extremely useful. Additional context may be provided by using a more rich task description and adding more instructions to the model input. Furthermore, post-processing can be performed by combining the inputs and outputs of two models, such as using the output from CodeT5 as input to GPT-3.5. Finally, bigger and more nuanced models such as GPT-4 can be explored to reveal whether the model can produce even better performance in Python code generation.

7 Individual Contributions

Originally, our group consisted of five members. However, Omneya Sultan had to withdraw from all classes due to personal issues. Despite this setback, we continued our project with the remaining members. Daniel focused on configuring VM instances on GCP and evaluating CodeT5 and CodeGen-2B using the MBPP dataset. Qasim evaluated GPT3.5 on the MBPP dataset, conducting both zero-shot and few-shot evaluations. Karen researched how to utilize the models and determine appropriate evaluation metrics. Saurav also contributed to evaluating the CodeGen2B model. All of the members worked on writing, editing and finalizing the report.

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *CoRR*, abs/2109.00859.

A Appendix

A.1 "No Result" Examples

A.1.1 GPT3.5

» Input: “Write a function to add the given list to the given tuples.”

» Output: “I am sorry, but your question is incomplete. Can you specify what the function should do with the list and tuples? Should it add the list as a new element to each tuple or should it add the values in the list to corresponding elements in the tuple? Please provide more information.”

A.1.2 CodeT5

» Input: Write a function to find the nth octagonal number. Your code should satisfy these tests:
assert is_octagonal(5) == 65
assert is_octagonal(10) == 280
assert is_octagonal(15) == 645

» Output:
assert is_octagonal(20) == 384
assert is_octagonal(25) == 516
assert is_octagonal(30) == 672
assert is_octagonal(

A.1.3 CodeGen

» Input:
Write a function to find sequences of lowercase letters joined with an underscore.
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match!')

» Output:
Write a function to find sequences of lowercase letters joined with an underscore.
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match!')
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match!')
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match!')
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match!')
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match!')
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match!')
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match!')
»Example: text_lowercase_underscore ("aab_cbbbc")==('Found a match

A.2 "FailedTest" Examples

A.2.1 GPT3.5

» Input: “Write a python function to convert a decimal number to binary number”

» Output:

```
def decimal_to_binary(n):
        binary = bin(n)
        return binary[2:]
```

A.2.2 CodeGen

» Input: "Write a function to find m number of multiples of n. Your code should satisfy these tests:"

assert multiples_of_num(4,3) == [3,6,9,12]
assert multiples_of_num(2,5) == [5,10]
assert multiples_of_num(9,2)== [2,4,6,8,10,12,14,16,18]

» Output:

```
def multiples_of_num(m, n):
    return [x for x in range(1, m+
                        1) if x % n =
                        = 0]
```

A.2.3 CodeGen

» Input: “Write a python function to find the volume of a triangular prism.”
»Example: find_Volume(10,8,6) == 240

» Output:

```
def find_Volume(a,b,c):
    return (a*b*c)/3
```