Code Summarization and Generation with Large Language Models: A Survey

MUHAMMAD HAROON, MICHAEL EWNETU, and MATTHEW DIRISIO, Michigan State University, USA

This paper explores the effectiveness of fine-tuning large language models for generating and summarizing code using different benchmark datasets. We implement a preprocessing pipeline to clean and tokenize data, apply sequence-to-sequence modeling techniques, and utilize evaluation metrics such as ROUGE [11], BLEU [14], and others for performance assessment. Experimental results demonstrate the model's efficacy in generating concise and coherent code and textual summaries, providing insights into automated code generation and summarization system improvements.

1 Introduction

Automated code summarization is crucial for improving software maintainability and developer productivity. Recent advancements in pre-trained sequence-to-sequence models, such as CodeT5[20], offer significant potential for this domain. This research project focuses on fine-tuning CodeT5[20] using the MBPP[1] dataset and the CodeXGlue dataset [12], leveraging its capabilities to understand Python code and generate human-readable summaries. Moreover, we explored fine-tuning CodeT5+, an improved version of CodeT5, on the CodeXGlue dataset. We outline the model's architecture, fine-tuning strategy, and evaluation framework to assess its performance in this critical application.

Jensen Huang, the CEO of Nvidia, believes that artificial intelligence could replace programming languages with English in the future. In order to accomplish that, models will have to tackle code generation, a fundamental challenge in automating software development. New models have become proficient at the task, driven by recent developments in large language models like the transformer architecture. This project also explores the application of transformer-based models and architectures like GPT to code generation. It measures their capability to understand and generate Python code by assessing generated code for syntactic and semantic similarities. We utilize the Py150 dataset and examine our model's architecture, training methodology, and evaluation metrics to evaluate our work for code generation.

For this paper, Muhammad Haroon worked with summarization with CodeT5 on MBPP, Michael Ewnetu worked with summarization with CodeT5 and CodeT5+ on CodeSearchNet, and Matt DiRisio worked with code generation with DistilGPT2 on Py150. All source code for this project can be found at https://github.com/haroon001/CodeSumGen.

2 Literature Review

2.1 Code Summarization

The field of code summarization has seen significant advancements, primarily driven by the adoption of neural and pre-trained language models. Traditional approaches relied on rule-based systems or statistical methods, which struggled to generalize across diverse programming languages and styles. Neural networks, particularly sequence-to-sequence architectures, brought improvements by learning mappings between code and natural language representations.

Early efforts such as CODE-NN [5] utilized Recurrent Neural Networks (RNNs) for code summarization, paired with attention mechanisms. While effective, these models were computationally expensive and prone to vanishing gradient issues with long code sequences.

The advent of transformers, exemplified by the T5 model [6], marked a turning point. Transformers eliminated recurrent connections, enabling efficient handling of long input sequences with attention mechanisms. The CodeBERT

Authors' Contact Information: Muhammad Haroon, haroonmu@msu.edu; Michael Ewnetu, ewnetumi@msu.edu; Matthew DiRisio, Michigan State University, East Lansing, Michigan, USA, dirisiom@msu.edu.

Manuscript submitted to ACM

model [8] adapted BERT for code-related tasks by pretraining on paired code and text datasets. While suitable for classification tasks, its architecture was less optimal for generative tasks like summarization. GraphCodeBERT [9] enhanced code representation by integrating program structure using Abstract Syntax Trees (ASTs). However, its complexity limited scalability for larger datasets.

OpenAI's Codex and similar GPT-based models [3] fine-tuned on code-specific data offered strong generative capabilities. Unfortunately, these models often require extensive computational resources. CodeT5 [20], based on the T5 transformer [6], excels in bidirectional sequence modeling, which is critical for understanding code dependencies. Unlike generic models, CodeT5 [20] incorporates both token-level and span-level denoising tasks, making it adept at handling incomplete or noisy code. CodeT5 [20] supports diverse code-related tasks, including summarization, classification, and translation, making it highly adaptable. By leveraging pre-trained knowledge, CodeT5 [20] achieves high performance with fewer resources compared to GPT-based models [3].

Moreover, CodeT5 is ideal for our experimentation as it solves a challenge faced by pre-trained models for Natural Language (NL) like BERT and GPT on Programming Language (PL) tasks. These models either rely on an encoder-only, in the case of BERT, or decoder-only, in the case of GPT, architecture, and pre-training, resulting in substandard performance in generation tasks. Moreover, they tend to process code snippets the same way as NL, not fully comprehending the unique characteristics of PL, including token types. The difference with CodeT5 is the unified pre-trained encoder-decoder transformer model architecture that utilizes code semantics from the developer-assigned identifiers. Hence, it employs a unified framework to support both code generation and understanding [20].

Additionally, the development of CodeT5+ demonstrated strong performance in real-world coding tasks. It builds upon its predecessor, CodeT5, and incorporates advanced techniques such as span denoising, text-code matching, and contrastive learning during pretraining to improve its understanding of both code and natural language. CodeT5+ supports multi-tasking and multimodal learning by unifying code and text through instruction tuning, making it adaptable to diverse scenarios like code generation, code summarization, and defect detection. The model is pre-trained on a large-scale dataset of 20+ programming languages and integrates knowledge from existing models such as CodeGen and PaLM, which enhances scalability and fine-tuning efficiency. The authors also introduce modular training strategies combining task-specific and general pretraining objectives. The evaluation shows that CodeT5+ achieves state-of-the-art results on benchmarks like CodeXGLUE, HumanEval, and MBPP. [7]

2.2 Code Generation

Code generation has seen similar progress in recent years. Probabilistic approaches to this task included the utilization of n-gram models or probabilistic context-free grammars to determine probabilities of generations. A particularly practical approach to code generation in this vein is PHOG, a generalization of a probabilistic CFG that can be trained just as effectively as a CFG while being more flexible and applicable to a broader range of programming languages [2]. Other approaches were more broadly statistical, estimating probabilities of a word appearing after the previous words in the sequence through training and learning probabilities from the training data. Some models taking this approach include n-gram language models and recurrent neural networks [17]. SLAMC was a statistical approach that augments n-gram models with semantic information in code tokens. With other improving techniques, the authors could generate code with a 64% accuracy, improving over other results [13].

This task was viewed as exceedingly difficult just 10 years ago, but now, models are becoming more and more proficient. This development was fostered mainly by the transformer architecture, which has led to a surge in the number of large language models developed over the past few years. The GPT models, Llama models, and Claude

models are three major players in the field recently, and the companies behind these models are constantly releasing new models that are effective at code generation of many languages.

3 Problem Description

We conducted experimentation and analysis in two areas of interest: Code summarization, where existing code is provided, and a natural language summary of the code is generated, and code generation, where incomplete code is provided and the continuation of the code is generated in its programming language, at either a token level or a line level. These tasks are both interesting individually but even more so together. Code summarization can help foster understanding of legacy code, accelerate problem-solving, and assist developers with documentation or comprehension. Code generation is a hot topic, with a seemingly endless flow of products and models being released to assist developers with code completion. Together, these tasks could a model's capacity to actually understand source code and the context of new code.

Automated code summarization is a sequence-to-sequence learning problem in which a code snippet is the input, and a concise natural language description of the code's functionality is the output. Furthermore, during fine-tuning, our model is trained auto-regressively using cross-entropy loss over a vocabulary set.

Code generation, also referred to as code completion or code-code generation, is a sequence prediction task where a model takes in partial code, or context, and outputs a predicted continuation of the code. This continuation can be either on a token level, a small unit of input that could be a word, a character, or just a sequence of characters, or on a line level, predicting a larger unit of code to achieve completion. As a result, both syntax and semantics of code are prevalent, differentiating this task from natural language generation due to distinctions between these languages. We trained a causal language model, diverging from the masked language model approach for summarization. In training, the model predicts subsequent tokens based on previous tokens and utilizes cross-entropy loss over the vocabulary of tokens. This allows the model to learn the syntax and semantics of code, like indentation, variable naming, scope, and bracket usage.

We took an interest in a couple of techniques used in model creation and training: knowledge distillation [10], which is utilized to create a relatively lightweight student model from a heavier teacher model, and transfer learning/fine-tuning, where pre-trained models are used, adapted, and retrained to fit a different task. In our case, we were especially interested in seeing how effective a distilled model could be at code generation after fine-tuning it on source code. This allowed us to meet time and resource constraints while leveraging large models that are already effective at text generation. For the reasons described below, we experimented with DistilGPT2 to determine how effective this model could be at token-level code completion.

4 Model Architecture

4.1 Code Summarization

T5 Transformer Architecture: The T5 (Text-to-Text Transfer Transformer) model [6] introduces a unified text-to-text paradigm, where all tasks are cast as text generation problems. Its core components include: The encoder processes the input sequence, producing a contextualized representation of tokens; The decoder generates the output sequence, conditioned on the encoder's representation; Each layer employs self-attention to compute token dependencies, enabling the model to capture long-range relationships; Position-wise feedforward layers, applied after attention, model non-linear transformations; T5 [6] is pre-trained using a "span corruption" objective, where random text spans are masked

and the model predicts the missing spans, promoting bidirectional context learning. CodeT5 [20] extends the T5 [6] architecture with modifications tailored to code tasks which include:

Multimodal Input Handling: Pretrained on a combination of natural language and code data (e.g., Python, Java, JavaScript), CodeT5 [20] learns to understand both syntax and semantics. It uses Byte-Pair Encoding (BPE) [4] tokenization methods to preserve code structure, such as indentation and delimiters.

Pre-training Objectives:

Masked Span Prediction: Masked spans within the code are predicted, similar to T5's objective, but optimized for code-specific features.

Identifier Prediction: Identifiers (e.g., variable names, function names) are masked and reconstructed, enhancing the model's capacity to understand programming context.

Code-Aware Denoising: Handles noisy or incomplete code by learning to infer missing segments, a common scenario in real-world programming.

Architectural Enhancements: Adjustments to the encoder-decoder settings allow better handling of programming languages, such as expanded token length for more extended code snippets. Fine-tuning for specific tasks leverages the pretraining knowledge, enabling rapid adaptation to code summarization.

4.2 Code Generation

Our model for this task utilized knowledge distillation [10] to create a lighter, faster version of GPT-2 [15] much like DistilBERT [19]. The result is a smaller model, DistilGPT2, that is more efficient and easier to run and work with without compromising much accuracy.

This model consists of auto-regressive decoder-only transformer blocks for sequential processes (as it is a causal LM that is trained to predict tokens that follow a given sequence), self-attention to focus on different elements of the input data, and other features like layer normalization and residual connections, like GPT-2.

GPT-2 was trained on the WebText corpus, containing millions of documents scraped from the internet, and DistilGPT2 was trained on a recreation of this corpus. As a result, the model was not trained on much code and is not proficient at generating code independently. For both ease of use and the model's lack of proficiency in generating code, the model was a prime target for us to apply transfer learning and fine-tuning to assess the effectiveness of these two techniques.

5 Methodology

5.1 Code Summarization

5.1.1 Code Summarization on MBPP Dataset.

Our dataset was the Mostly Basic Python Problems [1] problem set. This dataset had a Train-Validation-Test split of 38%, 9%, and 51%.

We preprocessed our data by clearing code and descriptions to remove extraneous characters and standardize formatting. Additionally, the model inputs were prefixed with "summarize:" to guide the model to our goal of summarizing the source code.

After the preprocessing, we tokenized the input data to send to the model. The source code and target descriptions were tokenized using the Byte-Pair-Encoding [4] method with a maximum sequence length (512 for inputs, 128 for outputs). Padding the inputs ensures batch consistency and that special tokens are handled appropriately.

For our training process, we loaded a pre-trained CodeT5 model [20] and fine-tuned it on the MBPP dataset described above using the Transformers package and the Hugging Face Trainer API. We used cross-entropy loss with label

smoothing and the AdamW optimizer with learning rate scheduling for 100 epochs. After training, the model's outputs were evaluated using ROUGE and BLEU calculations.

5.1.2 Code Summarization on CodeSearchNet (CodexGlue) Dataset.

CodeXGLUE is a robust benchmark for machine learning tasks related to code understanding and generation, providing a collection of datasets designed to evaluate models across diverse code-related tasks such as code summarization, code classification, and code generation. Among its included datasets is CodeSearchNet, which facilitates code-to-text search tasks by pairing code with relevant descriptions in natural language. This dataset is also particularly valuable for code summarization tasks, where models are trained to generate concise human-readable code summaries. Integrating CodeSearchNet into the CodeXGLUE project fosters more comprehensive evaluations, enabling the development and comparison of models designed to understand and generate code.[12]

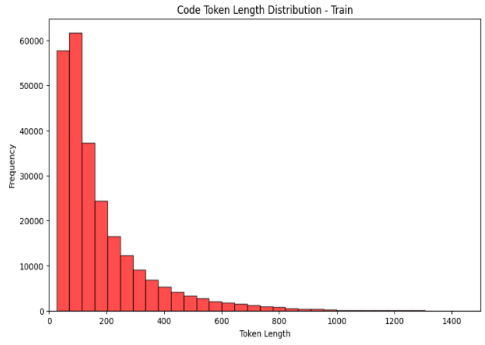

Fig. 1. Distribution of Input Token

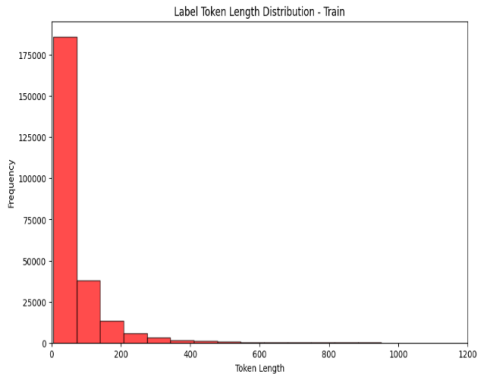

Fig. 2. Distribution of Output Token

Our work here only focuses on the Python portion of CodeXGlue. The preprocessing tasks included preparing the dataset for code summarization with the CodeT5 model in mind. First, the input code and corresponding docstrings are tokenized, ensuring both sequences are padded and truncated to fit within the model's maximum token lengths. As shown in Figure 1 below, the input token has an exponential distribution with few tokens having token lengths greater than 512, which is the maximum input length for the CodeT5 model. Hence, 512 was chosen as the max length parameter for truncation. Likewise, 128 was selected as the max length for the output token, which fits the distribution of output token length and the model requirements.

The code serves as the source sequence and input, while the docstring is treated as the target sequence for the summarization task output. Special handling is applied to replace padding tokens with -100 for accurate loss calculation during training. The dataset is split into training, validation, and test sets, with each set organized into batches for efficient model training.

In addition to tokenization, we applied a cleaning process to the input code where any existing docstrings are removed, ensuring that the model focuses only on the relevant code content. Again, we prefix the code to guide the model in summarizing the code. Then, the cleaned data is filtered to remove empty code or docstring entries, ensuring that only valid samples are included in the training process. Finally, the data is packaged into appropriate datasets and DataLoader objects, facilitating smooth and efficient training, validation, and testing workflows.

The fine-tuning process for CodeT5 and CodeT5+ was carried out using the Hugging Face Trainer API. The training involved setting up the training arguments with specific configurations, including a batch size of 16, a learning rate schedule with a warm-up step of 200, and a weight decay of 0.01. The number of epochs for CodeT5 was set to 2, while for CodeT5+, it was limited to 1 due to computational constraints. The evaluation strategy was set to "steps," with evaluation occurring every 2,000 steps, and the model's performance was monitored using the BLEU score as the primary metric for model selection. Additionally, model checkpoints were saved at every 4,000 steps, with a maximum of 5 checkpoints retained. The best model was automatically loaded after training based on the BLEU score, which guided the optimal model selection for code summarization.

5.2 Code Generation

We also utilized CodeXGlue as a starting point and a benchmark for this task. The authors describe the differences between token and line-level code generation, then provide a dataset and a model for the task [12].

We utilized the dataset they reference, Py150. This dataset was created for Raychev's 2016 paper, where they look to use this source code to train a decision tree for code completion and repair [16]. The data has an 80-10-10 split, but we restricted our usage to half of each subset for time and memory purposes. Thus, we used 60000 files for training and 7500 each for validation and testing.

The paper also presented a benchmark model, CodeGPT-small-py. This model is a transformer-based LM designed for code completion and text-to-code generation. It shares the GPT-2 architecture, containing 12 Transformer decoders, and is pretrained on Python code. It is pretrained from scratch, with parameters being randomly initialized. They also present a version of the model that initially uses GPT-2 parameters and is trained on the code corpus; this model is called CodeGPT-small-py-adaptedGPT2.

We conducted initial experimentation with this model but determined it would not be fruitful for several reasons. Namely, our fine-tuning process would likely not have much impact on the model given the large corpus of pretraining data, meaning we would not see much improvement in results. Additionally, we were more interested in applying transfer learning and fine-tuning to a model that was not initially trained on a lot of source code. As a result of both of these concerns, we transitioned instead to experimenting with DistilGPT2.

Tokenizing our data was accomplished with the corresponding tokenizer for the model, the GPT-2 tokenizer described above for code summarization. To load our model and this tokenizer we again utilize Hugging Face and the Transformers package to load our model and this tokenizer. We set the max length for our samples at 512, truncating or padding as needed. We used the end-of-sentence token as the padding token. We trained for three epochs with a $5x10^{-5}$ learning rate. We have 1000 warm-up steps, batch size 4, and gradient_accumulation_steps set to 8. We evaluated our model every 500 steps and chose the best model at the end of training to minimize evaluation loss.

We made predictions on the samples in our test set to evaluate the model. We randomly cut every sequence between the first and last few tokens depending on how many we predicted, sliced our data lists to reflect this random cutoff, and generated a prediction for the input sequence. We evaluated the generated code with a few metrics: BLEU, ROUGE, Average Levenshtein edit distance, and CodeBLEU.

BLEU is a standard metric for code generation, measuring the similarity between translated text (or, in this case, the generated code and the ground truth). However, BLEU is not without its weaknesses. The metric is designed to assess natural language and does not capture all of the syntax and semantics inherent to programming language, which is incredibly distinct from natural language. As a result, high BLEU scores for code can be misleading and are not always indicative of excellent translations for code as they would be for natural language.

Thus, we also analyze our model with CodeBLEU, a metric that captures more of the nuance of code. This metric, introduced in 2020, is more in line with programmer evaluations of similarity and is likely a better representation of the similarity between our generated code and the labels with which we assess similarity to [18].

Beyond that, we examine exact-match accuracy, which is especially prevalent and revealing for single-token code completion. This is the most important metric as it directly measures the model's effectiveness at generating code. There are more options to evaluate line-level code completion models, which we also present in our results below. These metrics, namely BLEU, ROUGE-1, and the Levenshtein distance/edit distance, measure the similarity between our generated code and our ground truth labels in various manners.

Lastly, we examine our model's beam predictions and use the top k predictions to calculate a form of pass@k defined by Chen et al. [3]. This form of the metric predicts the probability of ground truth appearing in the top k beam predictions by counting up occurrences where it does and dividing it by the total number of samples. For this metric, we generated longer sequences instead of a single token, hence some lower scores.

Training for this model was conducted on Google Colab, MSU HPCC, and a private GPU server.

6 Results

6.1 Code Summarization

Model	ROUGE-1	ROUGE-2	ROUGE-L	BLEU
CodeT5	0.72	0.55	0.70	0.46

Table 1. CodeT5 fine-tuning results for MBPP benchmark.

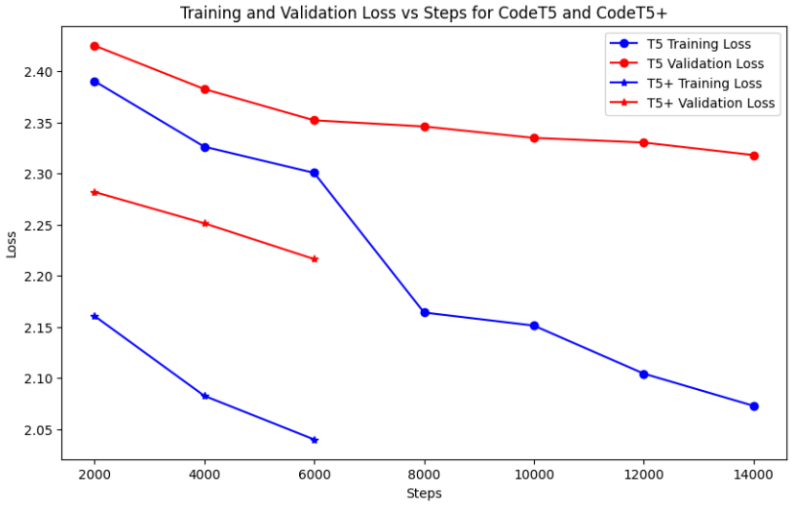

Fig. 3. Loss vs Step for Fine-tuning CodeT5 and CodeT5+ on CodeSearchNet (CodexGlue) dataset

Model	ROUGE-1	ROUGE-2	ROUGE-L	BLEU
CodeT5	0.474	0.192	0.419	0.251
CodeT5+	0.488	0.206	0.439	0.266

Table 2. CodeT5 and CodeT5+ fine-tuning results for CodeSearchNet (CodeXGlue) benchmark.

6.1.1 Fine-tuning on MBPP Benchmark Examples.

Example 1 Original Code:

```
R = 3
C = 3
def min_cost(cost, m, n):
    tc = [[0 for x in range(C)] for x in range(R)]
    tc[0][0] = cost[0][0]
    for i in range(1, m+1):
        tc[i][0] = tc[i-1][0] + cost[i][0]
    for j in range(1, n+1):
        tc[0][j] = tc[0][j-1] + cost[0][j]
    for i in range(1, m+1):
        for j in range(1, n+1):
            tc[i][j] = min(tc[i-1][j-1], tc[i-1][j], tc[i][j-1]) + cost[i][j]
    return tc[m][n]
```

Original Description: Write a function to find the minimum cost path to reach (m, n) from (0, 0) for the given cost matrix `cost[][]` and a position (m, n) in `cost[][]`.

Generated Summary: Write a function to find the minimum cost of a pair of integers for the given triangular matrix.

Example 2 Original Code:

```
def similar_elements(test_tup1, test_tup2):
    res = tuple(set(test_tup1) & set(test_tup2))
    return (res)
```

Original Description: Write a function to find the similar elements from the given two tuple lists.

Generated Summary: Write a function to find the similar elements in the given two tuples.

Example 3 Original Code:

```
import math
def is_not_prime(n):
    result = False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            result = True
    return result
```

Original Description: Write a Python function to identify non-prime numbers.

Generated Summary: Write a function to check whether the given number is not prime or not.

6.1.2 Fine-tuning on CodeSearchNet(CodexGlue) Benchmark Examples.

The following contains generated output for direct evaluation from the two models, CodeT5 and CodeT5+, fine-tuned on the CodeSearch(CodexGlue) dataset. The inputs for the four cases are all Python functions for computing binary search. However, the function's name is changed for the three cases to learn where the model is fixing its attention.

Example 1 Original Code:

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (high + low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Code T5 Generated Summary: Binary Search

Code T5+ Generated Summary: Binary search.binary_search([1, 2, 3, 4, 5...

Example 2 Original Code:

```
def fun_1(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (high + low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Code T5 Generated Summary: Return the index of the first element in the array that is equal to x

Code T5+ Generated Summary: Function to find the midpoint of an array that is equal to x.

Example 3 Original Code:

```
def factorial(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (high + low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Code T5 Generated Summary: Return the index of a factorial value in the array
Code T5+ Generated Summary: Return the factorial of an array

Example 4 Original Code:

```
def bubble_sort(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (high + low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Code T5 Generated Summary: Bubble sort
Code T5+ Generated Summary: Bubble sort. bubble sort([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

6.2 Code Generation

Before fine-tuning, we loaded DistilGPT2 from Hugging Face and evaluated it on our test set of 7500 to get a starting point for this model's performance. The results are in Table 3.

# of Tokens Generated	Accuracy	BLEU	ROUGE-1	Average Levenshtein Distance	Average Length	CodeBLEU
1	47.9%	0.1791	0.1861	2.1063	2.9	0.4367
3	20.7%	0.1772	0.2094	6.7950	8.7	0.1774
5	11.8%	0.1378	0.1977	11.6489	14.4	0.1593

Table 3. DistilGPT2 Results Without Fine-Tuning

After that, we fine-tuned the model on our train set and assessed it on the same test set to determine how effective our transfer learning application was for this model. The results follow in Table 5.

# of Tokens Generated	Accuracy	BLEU	ROUGE-1	Average Levenshtein Distance	Average Length	CodeBLEU
1	70.6%	0.7716	0.3070	1.1479	2.9	0.4668
3	48.1%	0.4902	0.4075	3.9061	8.7	0.2964
5	36.7%	0.4146	0.4147	7.0340	14.4	0.3472

Table 4. DistilGPT2 Results With Fine-Tuning

Lastly, we generated k beam predictions and utilized them to calculate pass@k. The results follow in Table 5.

k	Value
1	0.377
3	0.443
5	0.470

Table 5. DistilGPT2 pass@k Scores With Fine-Tuning

7 Conclusions

For our code summarization task on the MBPP dataset, our model's generated summaries were closely aligned with human-written descriptions, demonstrating the model's effectiveness in understanding and explaining Python code. ROUGE-1 (0.72) measures unigram (single word) overlap between the generated and reference summaries. This score of 0.72 indicates high content similarity and suggests the model correctly captures about 72% of individual words. ROUGE-2 (0.55) measures bigram (two-word phrase) overlap. This score of 0.55 suggests moderate performance in preserving exact word sequences and indicates some difficulty in maintaining precise phrase-level coherence. ROUGE-L (0.70) measures the longest common subsequence between generated and reference summaries, and our score of 0.70 suggests high structural similarity while indicating that the model maintains good overall sentence structure. Our model's BLEU score of 0.46 is considered moderate to good in many text generation tasks, including code summarization. It suggests the generated summaries have a reasonable level of overlap with reference summaries but are not necessarily perfect. We are happy with our model's performance on this task.

On the other hand, in the code summarization part of the CodeXGlue dataset, the results were moderate. The ROUGE-1 score from the evaluation results for CodeT5 indicates that CodeT5 can match a significant portion of unigram overlaps between the generated and reference summaries, with a score of 0.474. The ROUGE-2 score, at 0.192, suggests that CodeT5's performance in matching bigrams is relatively lower, implying that it struggles more with generating syntactically complex phrases. The ROUGE-L score of 0.4198, which considers the longest common subsequence, indicates that CodeT5 is reasonably good at preserving the overall structure of the reference summaries but may miss some finer details. The BLEU score of 0.25 suggests that while CodeT5 generates summaries with some degree of accuracy regarding token-level precision, there is still substantial room for improvement, especially in terms of fluency and precision when compared to the references.

For CodeT5+, the increase in the ROUGE-1 score to 0.488862 indicates that CodeT5+ is better at capturing unigram overlaps, improving the relevance of the generated summaries compared to CodeT5. The ROUGE-2 score of 0.206560 represents a noticeable improvement in the model's ability to generate bigrams, suggesting that CodeT5+ is more adept at generating phrases that align with the reference summaries. The ROUGE-L score of 0.433068 reflects a slight improvement in preserving the overall structure of the reference summaries, highlighting CodeT5+'s enhanced ability to generate more coherent and contextually accurate summaries. Finally, the BLEU score of 0.266115 shows that CodeT5+ generates summaries with better token-level precision than CodeT5, offering a more fluent and precise output.

Moreover, direct evaluation gave insight into the limitations of the models that may not be captured through the ROUGE and BLEU metrics. For instance, as shown in the result section, the models seem to fixate on the name of the function. When the function's name is changed to uninformative text, like "fun_1" or "helper_function," the models attempt to describe the function by focusing on the body part. However, suppose the function's name is changed to a misleading name with a closer application, for instance, renaming binary search to bubble sort. In that case, the models disregard the body part and fixate on the function's name. This problem can be better analyzed by visualizing where transformer models like CodeT5 and CodeT5+ are putting their attention.

We also determined that fine-tuning DistilGPT2 with our training set saw marked improvements in every metric across the board. Our primary interest was in single token exact-match accuracy for token-level prediction, and the model saw an increase of almost 23% in this metric. This wasn't the only metric boosted by our utilization of fine-tuning—all of our metrics saw a jump when we evaluated the fine-tuned model compared to the base version. This boost is accentuated as we generate longer sequences, which is more complex than generating a single token.

For our fine-tuned model, we obtain an excellent BLEU score (.7716), indicative of a great degree of similarity between our generated code and the ground truth. When examining CodeBLEU instead, we see that the generated code is still a good translation of the ground truth, while not as good as our BLEU score. Our BLEU, ROUGE-1, and CodeBLEU scores are representative of a good translation between the two snippets of code, and we are confident in our model's ability to generate Python code given this information.

We were pleased with our pass@k scores as well. They indicate that our model can generate a correct prediction for a more extended sequence almost half of the time, somewhere in the top 5 beam predictions. This suggests that the model is not necessarily ineffective for longer sequences as exact match accuracy trends that way. It also indicates that our model could use further tuning and training to improve.

Our experimentation with the GPT architecture and a distillation of the GPT-2 model was engaging. These results exemplify the effectiveness of transfer learning and fine-tuning as it relates to code generation while also exploring the impacts of knowledge distillation. We can conclude that the fine-tuning process for our project was a success, resulting in noticeable performance improvements in every metric.

8 Future Work

For code summarization, we would like to expand our dataset and training process to include diverse source code, allowing our model to generalize better and summarize a broader range of programming languages. We also would like to integrate additional metrics, such as METEOR and CIDEr, to evaluate our model holistically. Additionally, we would like to investigate our model's interpretability to understand the situations and samples where it fails to summarize code effectively, and we hope to explore larger-scale CodeT5 models/ensemble approaches in the future for enhanced summarization performance.

In the future, we also have plans for the code generation aspect of our project. Our first goal is to train the model on more code. Our model and training process were limited somewhat by time and memory constraints, which drove us to use only a subset of the Py150 dataset for training and fine-tuning. With access to more resources, we could train the model for longer and on a more significant portion of the data, if not all of it. This would hopefully increase the token vocabulary of our generation models, theoretically allowing it to predict a wider range of tokens and improve its generation ability.

Second, we plan to explore more model architectures to gain a more comprehensive understanding of the nuances of a broader suite of models as they concern token-level code completion. We achieved a solid understanding of GPT-2 and its distilled version as they relate to token-level code completion, as well as the techniques and impacts of fine-tuning/transfer learning. Still, we would like to experiment with more models. Our primary considerations at this time are Codex [3] and CodeT5, the latter of which would give more cohesion to the two aspects of our project.

Lastly, while we did experiment with generating longer sequences of tokens, this is not necessarily the same task as line-level completion, which is only becoming more prevalent in development environments. While we are satisfied with our results for token-level completion, we hope to explore line-level completion in the future, ideally obtaining a model that is effective at both tasks instead of one or the other.

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, and Chris Olah. 2021. The MBPP Dataset: Mostly Basic Python Programming Problems for Learning and Benchmarking. https://arxiv.org/abs/2108.07732

[2] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.). PMLR, New York, New York, USA, 2933–2942. https://proceedings.mlr.press/v48/bielik16.html

[3] 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. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[4] Alec Radford et al. 2019. Language Models are Unsupervised Multitask Learners. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[5] Iyer et al. 2016. Summarizing Source Code using a Neural Attention Model. https://aclanthology.org/P16-1195/

[6] Raffel et al. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. https://arxiv.org/abs/1910.10683

[7] Yue Wang et al. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. https://arxiv.org/abs/2305.07922

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Lin Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://aclanthology.org/2020.emnlp-main.594

[9] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Nan Duan, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *Proceedings of the 2021 International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=jLoC4ez43PZ

[10] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. arXiv:1503.02531 [stat.ML] https://arxiv.org/abs/1503.02531

[11] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[12] 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. arXiv:2102.04664 [cs.SE] https://arxiv.org/abs/2102.04664

[13] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 532–542. https://doi.org/10.1145/2491411.2491458

[14] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[15] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[16] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *ACM SIGPLAN Notices* (2016), 731–747.

[17] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. *SIGPLAN Not.* 49, 6 (June 2014), 419–428. https://doi.org/10.1145/2666356.2594321

[18] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

[19] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv:1910.01108 [cs.CL] https://arxiv.org/abs/1910.01108

[20] Yue Wang, Xiaodong Hu, Shafiq Joty Shi, and Yulia Tsvetkov. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://aclanthology.org/2021.emnlp-main.685/