Pseudocoder: An Analysis of Various Architectures on Python Code-To-Pseudocode Translation

Wenda Gu
wendagu@usc.edu

Egor Cherkashin
cherkash@usc.edu

Sarah Chen
snchen@usc.edu

Abstract

It is well-known that pseudocode is useful for learning purposes and comprehension of existing code. However, pseudocode has yet to be considered for applications beyond code comprehension, such as debugging functionality. Theoretically, an accurate code-to-pseudocode translation of broken code can easily reveal or rule out logical flaws within the program, expediting the debugging process. In this paper, we use the methods of transformers, RNNs with and without attention, and prompting and fine-tuning existing LLMs to approach Python code-to-pseudocode translation. By approaching the problem with varying solutions, we aimed to find the benefits of each architecture and determine an optimal approach to the code-to-pseudocode translation problem. Ultimately, we found that [more about results here].

1 Introduction

1.1 Motivation

Having a reliable model for code-to-pseudocode translation enhances clarity, collaboration, and learning in software development environments. It aids in comprehension and documentation of complex algorithms or pieces of code by providing a clear and concise representation in natural language. This simplification is especially helpful for individuals who may not be proficient in a particular programming language but need to understand the logic behind the code. Additionally, it facilitates collaboration among developers by bridging the gap between different programming backgrounds, allowing team members to communicate and review code more effectively. Automated translation to pseudocode can serve as a valuable educational tool, helping students grasp programming concepts by breaking down code into more easily understandable steps.

Additionally, there is great potential for pseudocode to be useful for debugging purposes. Especially for those who outline their code with some general pseudocode, having the ability to translate their code to pseudocode could help reveal logical erros within the code when compared against the intended functionality.

1.2 Task Definition

Our language models will receive Python code as input and produce pseudocode as output. See Table 1 for an example. We will have 3 separate models: an RNN with and without attention, transformers, and prompting and fine-tuning an existing large-language model. This comparative analysis aims to shed light on how each model navigates the challenges of translating code into pseudocode. It is not our intention to seek a singular "best" model but to understand the relative strengths and practical applications of these models in the domain of code translation.

2 Related Work

There is a significant amount of research on the code-to-pseudocode and pseudocode-to-code problem. With Python specifically, one of the largest Python code to pseudocode datasets, Django, was created for the research paper Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (Oda et al., 2015). Other Python code to pseudocode models have been created and researched, but are moreso line-by-line translations and lack explanations of the purpose of the code. Additionally, the research paper "SPoC: Search-based Pseudocode to Code" (Kulal et al., 2019) performed C++ code-to-pseudocode translation. Especially notable is their concession that BLEU may not be comprehensive enough of a metric for code-to-pseudocode translation, which we will discuss later.

Input	Output
`def sort(arr):`	`in function sort, which takes an array of integers arr`
`  n = len(arr)`	`  let n be the length of arr`
`  for i in range(n):`	`  for every i in [0, n)]:`
`    for j in range(0, n - i - 1):`	`    for every j in [0, arr-i-1):`
`      if arr[j] > arr[j+1]:`	`      if value of arr at j > value of arr at j+1`
`        arr[j], arr[j+1] = arr[j+1], arr[j]`	`        swap value at j with value at j+1`
	`Explanation: Implementation of Bubble Sort Algorithm.`

Table 1: Example input and output for our LLM.

3 Hypothesis

A specially trained language model for code to pseudocode translation and explanation will perform better than general purpose language models.

4 Datasets

In this project, we will leverage two primary datasets: the Django Dataset and the CoNaLa (Code Natural Language) Dataset. Both datasets consist of Python code snippets paired with their corresponding English pseudocode annotations, which are crucial for training our model to understand and generate accurate pseudocode and explanations.

4.1 Django Dataset

The Django Dataset was originally created for the research paper "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation" (Oda et al., 2015), which was discussed in the "Related Work" section. It consists of 18,805 annotations, divided into 16,000 training, 1,000 development, and 1,805 test annotations. Each annotation consists of a line of Python code and its corresponding manually written natural language description. For ease-of-use, we specifically used the Django dataset on Hugging Face by AhmedSSoliman, as the original Django dataset in the Github repository is written as two text documents (one of code and another of pseudocode). By using this Hugging Face dataset, we can expedite our process by removing the need to parse and categorize the data ourselves.

4.2 CoNaLa Dataset

The CoNaLa Dataset, or The Code/Natural Language Challenge dataset, was developed by two Carnegie Mellon University labs to provide a corpus for code generation from natural language descriptions (Yin et al., 2018). The dataset crawled from Stack Overflow and curated by annotators contains 2,879 total examples, split into 2,379 training and 500 test examples. While a larger dataset of 600,000 automatically-mined code examples is also available, it is not filtered and does not contain proper code-to-pseudocode annotations for each example, which is necessary for our purpose.

4.3 Creating a Single Dataset

Given the structural similarity between these two datasets—where each data point is a line of Python code matched with an English pseudocode annotation—we plan to combine them to enhance the diversity and volume of our training material. This merged dataset will provide a more robust foundation for developing our language model, enabling it to handle a wider range of code-to-pseudocode translation tasks.

To begin, we adopted a (90/5/5) data split for training, validation (inferencing), and testing purposes. Accordingly, we allocated 18,300 examples for training, 1,140 for validation, and another 1,141 for testing. This distribution aims to ensure comprehensive coverage and effective learning while leaving room for adjustments.

4.4 Tokenization

To tokenize the data, we use two Hugging Face BPE (Byte-Pair Encoding) tokenizers for Python code and pseudocode respectively. The tokenizers are trained on their respective complete datasets, comprising the training, validation, and test data from both the Django and CoNaLa datasets for Python code or pseudocode, with the following special tokens: `"[UNK]"` for the unknown token, `"[BOS]"` for the beginning-of-sentence token, `"[EOS]"` for the end-of-sentence token, `"[PAD]"` for the padding token. A maximum length of `70` is set for the tokenizers for both truncation and padding, based on the average sentence length in the dataset.

5 Evaluation Framework

Our project adopts a comprehensive evaluation framework designed to assess the performance of various models in code-to-pseudocode translation. This framework is structured to provide insights into how effectively each model handles the task, focusing on accuracy, linguistic fidelity, and the models' ability to generalize across different code snippets. Given the exploratory nature of our research, our evaluation criteria are geared towards understanding the strengths and areas for improvement of each model rather than establishing a hierarchical ranking.

To test the models' generalization capabilities, we will evaluate their performance across a diverse set of code examples, including those with varying levels of complexity and from different domains. This will help us understand how well the models can adapt to new, unseen code snippets.

Our evaluation framework consists of 3 evaluation metrics: BLEU, ROUGE, and Semantic Similarity.

5.1 BLEU Scores

BLEU, or Bilingual Evaluation Understudy, is an algorithm for evaluating the acccuracy of text translation from one natural language to another by using n-grams (Papineni et al., 2002). Addressed in "SPoC: Search-based Pseudocode to Code" (Kulal et al., 2019), BLEU may not be the best metric for evaluating code-to-pseudocode translation because it fails to account for whether or not code is functionally correct. However, upon further analysis of related work on LLM code-to-pseudocode translation, we found that most research on this problem used BLEU as an evaluation metric. Thus, while BLEU may not be a comprehensive evaluation of translation quality, we thought it still appropriate to include due to its usage in related work, as it would allow us to compare our results with that of other research.

5.2 ROUGE Scores

ROUGE, or Recall-Oriented Understudy for Gisting Evaluation, is another set of metrics that is used to evaluate text summarization tasks by looking at overlapping n-grams between generated and reference texts (Lin, 2004). We found it appropriate to calculate ROUGE scores because the code-to-pseudocode translation task is not just a machine translation task but also a summarization task. To elaborate, unlike natural language to natural language translation tasks, there is no one "correct" translation. Therefore, we include ROUGE Scores to evaluate the summarization capabilities of our models when translating code-to-pseudocode.

5.3 Semantic Similarity

Finally we use Semantic Similarity, a metric that uses the similarity between concepts and terms from prior knowledge sources to estimate the similarity between text (Slimani, 2013). Our two other metrics, ROUGE and BLEU, use n-grams to evaluate the quality of generated text against the reference or ground truth text. While this is useful, as discussed earlier, we know that in the code-to-pseudocode translation problem, a translation does not have to use the same words or phrases as the ground truth to still be an effective and correct translation. Previous research has tried to address this problem with various approaches. For instance, in "SPoC: Search-based Pseudocode to Code" (Kulal et al., 2019), Kulal et al. created their own evaluation framework to evaluate the functional correctness of translations. In "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation" (Oda et al., 2015), Oda et al. had two forms of human evaluation, from both experienced and beginner programmers, to evaluate both comprehensiveness and readability of generated translations. However, methods such as these require resources beyond our scope, and thus we attempted to address this problem in a simpler manner, by calculating Semantic Similarity.

6 Approach

In attempt to identify the strengths of particular architectures in the code-to-pseudocode translation task, we employed 3 main approaches: Recurrent Neural Networks (RNNs) with and without attention, transformers, and fine-tuning and prompting an exsisting pre-trained model, CodeLlama.

6.1 Recurrent Neural Networks (RNN) with and without Attention

Given their efficacy in handling sequential data, RNNs represent a fundamental approach for our task. We aim to explore their capacity to capture the temporal dependencies characteristic of code syntax and structure.

6.2 Transformers

Given their state-of-the-art performance in various natural language processing tasks, including translation, we anticipate that Transformer models will offer significant improvements in translating code to pseudocode. Their self-attention mechanisms can handle long-range dependencies more effectively than conventional RNNs, potentially leading to more accurate and coherent translations.

6.3 CodeLlama

After our creation and evaluation of the prior two models, we quickly realized that our dataset was not nearly robust enough to enable the models to properly comprehend code. Therefore, we wanted to perform similar evaluations on a model pre-trained to understand code and code-related language, and compare its efficacy to that of our custom models.

As a model specifically designed for programming language tasks, CodeLlama presents an intriguing option for fine-tuning. We expect its pre-existing knowledge of code structures and syntax to be beneficial for our code-to-pseudocode translation task.

7 Experiments and Results

7.1 RNNs

The RNN model used for this task is an encoder-decoder model where both the encoder and decoder are RNN networks. The intuition for using this encoder-decoder structure is that this is a translation task between two languages, in which the source and the target vocabulary are different. Therefore, the encoder takes the source vocabulary (Python code) into concern when encoding, while the decoder calculates the probabilities over the target vocabulary (pseudocode).

In this part of the experiment, the question is that whether the use of attention mechanism will improve the performance of the RNN model. The answer is yes, but the improvement is not significant. Our speculation is that since the sentences in the translation task between Python code and pseudocode are short, the power of attention mechanism is not fully demonstrated. Moreover, because the insignificant difference between the two models, the additional time added by attention layers during training and inference time may not be worth it.

7.2 Transformers

In our experiments, we specifically focused on testing how the size of embeddings and feedforward neural networks (ff neural nets) within the Transformer architecture affect performance. Our results confirmed that increasing the size of embeddings consistently improves the model's performance. This enhancement likely stems from the model's increased capacity to capture and process a wider array of syntactic and semantic nuances present in the source code.

Conversely, augmentations to the size of the feed-forward neural networks did not yield significant changes in performance. This suggests that once a sufficient capacity to handle the sequential logic of code is reached, simply expanding the ff neural nets does not contribute additional benefits. This finding aligns with the hypothesis that the critical factors for code-to-pseudocode translation lie more in capturing relationships and dependencies than in processing capacity alone.

Moreover, our testing with different generation strategies revealed that greedy generation often resulted in degenerate behavior, characterized by repetitive or irrelevant pseudocode outputs. In contrast, temperature-based sampling produced more varied and contextually appropriate translations. This discrepancy likely arises from the limited vocabulary size and the insufficient training dataset size used in our experiments. The greedy approach's tendency to favor higher probability sequences exacerbates these limitations.

To illustrate, consider the following example comparing temperature-based sampling and greedy generation outputs:

Another interesting observation was the heavy influence of variable names on the translations produced by our Transformer models. This phenomenon indicates that our models, which have not undergone pre-training specific to programming languages, are overly reliant on direct associations between variable names and their usage contexts. This suggests a potential area for further research and model refinement, possibly through pre-training on a more diverse coding corpus to reduce the models' sensitivity to variable naming and improve their generalization capabilities.

7.3 CodeLlama

For prompting and fine-tuning, we specifically used CodeLlama-7b-Instruct, part of Meta's Code Llama

Sampling Output
return a list of the object ‘to self.py‘ lst ‘to object‘ y ‘to self.py‘ of the self.loader to a list of the object ‘to a list of the object‘ that ‘in descending to string
Greedy Output
return a list of the class ‘to a list of the class‘ using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using ‘using

Table 2: Comparison of pseudocode outputs using temperature sampling (t = 0.5) and greedy generation strategies.

family of pretrained and fine-tuned generative text models with varying parameter sizes and purposes. CodeLlama-7b-Instruct is a CodeLlama Instruct model trained on 7 billion parameters and fine-tuned to perform instruction and chat-based tasks. While the Code Llama Python models also exist, we chose this model over the other models in the CodeLlama family because the Instruct model would be able to take in our custom prompts for prompting evaluation. For fine-tuning, we continued to use CodeLlama-7b-Instruct for consistency and accuracy of comparison between prompting and fine-tuning.

Due to computational limitations, CodeLlama-7b-Instruct was fine-tuned only on the Django dataset. However, both fine-tuned and prompted generations were evaluated based on their generations on both the Django and CoNaLa test sets.

7.3.1 Prompting CodeLlama

In evaluating CodeLlama-7b-Instruct's ability to perform code-to-pseudocode translation with prompting, we took two approaches: zero-shot prompting and few-shot prompting. In both approaches, we modified the prompt until the model generated the most desirable translation, based on both our evaluation metrics and human evaluation.

For zero-shot prompting, the optimal prompt we found followed the format:

```
"[INST]\nWrite the following Python code
as one line of natural language
pseudocode\n" + code + "\n[/INST]"
```

where the [INST][/INST] tags are the tags used by CodeLlama to differentiate user input and generated output, and code is the line of Python code to be translated. Using this prompt format, we ran all lines of code in our test set through CodeLlama-7b-Instruct, then ran our evaluation metrics on the generated translation against our test set of human-annotated pseudocode translations. See Table 3 for results.

For few-shot prompting, the optimal prompt we found followed the format:

```
"""[INST]
Write the following Python code as one line
of pseudocode:
s.split(' ', 4)
[/INST]
Split a string `s` by space with `4` splits
[INST]
Write the following Python code as one line
of pseudocode:
text.split()
[/INST]
split string `text` by space

Write the following Python code as one line
of pseudocode:
<code>
[/INST]"""
```

where the [INST][/INST] tags are the tags used by CodeLlama to differentiate user input and generated output, and <code> is the line of Python code to be translated. Similarly to zero-shot prompting, we ran all lines of code in our test set through CodeLlama-7b-Instruct with the above prompt, then ran our evaluation metrics on the generated translation against our test set of human-annotated pseudocode translations. See Table 3 for results.

7.3.2 Fine-Tuning CodeLlama

We utilized PEFT (Parameter-Efficient Fine-Tuning) and SFT (Supervised fine-tuning) to fine-tune CodeLlama-7b-Instruct. As mentioned previously, computational limitations placed limitations on our ability to fine-tune CodeLlama-7b-Instruct. Using PEFT helped us train our model efficiently, though we also had to reduce some parameters.

CodeLlama-7b-Instruct was fine-tuned on the Django dataset train split with the following parameters:

```
    learning_rate = 5e-4
    train_epochs = 1
```

```
save_steps = 500
logging_steps= 250
per_device_train_batch_size=2
```

To compare the performance of the fine-tuned model against prompting, we used the exact same zero-shot and few-shot prompts described earlier but on our fine-tuned model. See Table 3 for results.

As expected, Few-Shot prompting performed better than zero-shot prompting, in both pure prompting and fine-tuning. Surprisingly, though, while fine-tuned zero-shot had the best ROUGE-L score, it had significantly lower BLEU and Semantic Similarity scores. However, this could be a result of our scaled-down fine-tuning, which was only one epoch and had a very minimal batch size, which could have effected the quality of fine-tuning. This could also be the case for few-shot fine-tuning, but we were unable to obtain ROUGE scores for this test case due to some empty outputs.

8 Conclusions

Although this is a perfect use case of the encoder-decoder model in machine translation, the result is not ideal, which is probably due to the nature of the task itself and the small dataset size.

First, although the task is a translation task, it is not a typical natural language to natural language translation task. It deals with Python code, which, empirically, is more similar to English than other programming languages, and pseudocode is represented by natural language but the actions that it describes are highly structural and relate highly to the programming language. These characteristics of our source and target languages are the probable causes to the poor performance with our traditional approaches to machine translation

Second, the small dataset size limits the ability for our models to generalize, as there are many variables like variable names, function names, error messages, etc. that confuses our model when the data is not sufficient.

9 Future Considerations

Informed by our findings, we may explore additional strategies to enhance model performance, including but not limited to:

1. Implementing advanced evaluation metrics for a more nuanced understanding of translation quality.
2. Exploring data augmentation techniques to improve model robustness and training efficiency.
3. Investigating hybrid models that combine the benefits of different architectures.
4. Finding ways to generalize variable names before translation to pseudocode, as various variable names can have equivalent meaning in code.

References

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. Spoc: Search-based pseudocode to code.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Annual Meeting of the Association for Computational Linguistics*.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 574–584, Lincoln, Nebraska, USA. IEEE Computer Society.

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*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Thabet Slimani. 2013. Description and evaluation of semantic similarity measures approaches. *International Journal of Computer Applications*, 80(10):25–33.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

	BLEU	**ROUGE-l**	**Semantic Similarity**
Prompting: Zero-Shot	0.1519	0.2646	0.7232
Prompting: Few-Shot	0.1651	0.3119	0.7481
Fine-Tuning: Zero-Shot	0.1223	0.3388	0.6239
Fine-Tuning: Few-Shot	0.1411	N/A	0.6393

Table 3: Evaluation metrics on outputs of CodeLlama-7b-Instruct