Pseudocode to Code Translation Using Transformers

Stanford CS224N Custom Project

Nazli Ugur Koyluoglu, Kaan Ertas, Austin Brotman
Department of Computer Science
Stanford University
`{nazli, kertas, abrotman}@stanford.edu`

Abstract

Pseudocode to code translation is an open field of research, with work impacting a variety of disciplines. We approach the problem by employing transformers for the task of pseudocode-to-C++-code translation, and do a comparative study with the earlier published results using LSTMs. We employ a pair of architectures, tokenizers, and make use of pretrained English language models to boost training. We also explore the impact of different types and amounts of context as input to our models. Using functional correctness for performance evaluation as opposed to traditional methods, our results match the performance of previous work closely, and point towards additional benefit of context in line-by-line translations.

Our project mentor is Lauren Zhu.

1 Introduction

An important nuance in the field of machine translation is that different translation tasks contain additional subtleties that need to be addressed by the model. Minor mistakes in translation can often be overlooked in target languages with less structure and rigidity, as the user can still grasp the semantic content of the sentence ("I going to the supermarket" will not trouble most native English readers about the sentence's semantic content). This is not the case for structured languages with strict rules, where a misplaced (or missing) token can result in the failure of the entire translation. `for (int i=0; i<5; i++)` is not equivalent to `for (int i=0; i<5; i+)` in the eyes of a compiler. Therefore, extra care must be taken in learning not only semantic content, but also precise syntax.

If we take English as the source language and a programming language as the target language, we define a neural machine translation task that falls under the latter category. While the source language is unstructured, the target language is highly structured; and for compiled languages, there is the additional benchmark of whether a program compiles at all, let alone whether it succeeds at runtime. Furthermore, the idea of coreference resolution (identifying all mentions that reference the same entity) is extremely important for this domain, as variable names need to be preserved across lines in the same scope so that the expected functionality is produced. The last hurdle is that not only do lines have to be internally consistent, they have to be able to replicate the functionality that is desired of the complete program.

This paper investigates pseudocode-to-code translation in the special case of English pseudocode to C++ code translation. In an attempt to make the translation task more tractable, a natural way of decomposing the problem is to consider line translations, and then build a program translation from separate line translations. Admittedly, treating line translations as atomic and independent of other lines ignores the problem of line incompatibility. The success of a line that increments a variable is contingent on the success of the line that initialized the variable. However, this can be treated as a search problem once we have multiple candidate translations for each line. In the interest of time, we focus on the problem of line translations, and we define our research question as follows: Given a line of pseudocode, can we produce a line of C++ code that is both compatible with the rest of the program and can pass test cases on the program level?

2 Related Work

Kulal et al. have released the SPoC dataset, a human-annotated line-by-line pseudocode-to-C++ code translation dataset with program-level test cases[1]. They frame the translation task as a search on two different levels: the line level and the program level. Line-level translations are performed through an LSTM encoder-decoder with attention, beam search, and coverage vectors. Taking a line of pseudocode as input, the model outputs a list of candidate translations produced by beam search with their accompanying probabilities predicted by the model. Program level translations take candidate translations for each line, and perform a search until either a certain budget is reached or a succeeding program has been reached. The search is informed by compilation errors and test case runs. Since the line that results in the compilation error is not always the first line in the program that causes the issue, the authors employ different selection methods for the problematic line; while one method involves the use of an additional LSTM for problematic line prediction, another iterates to find the first prefix of the generated program that doesn't compile and continues search from that point.

The authors also introduce metrics of functional correctness, which deviates from the generally employed metrics such as BLEU scores. Such metrics are all the more important for coding languages as opposed to natural languages, since functional incorrectness in C++ will either lead to outright compilation errors, or failed test cases. The authors highlight that another major challenge regarding pseudocode to code translation involves translations that are correct on a line-by-line basis being incompatible in a full program. This is emphasized by their success of 84-87% on individual lines but only 24.6% on full programs when using direct translation.

The data from this paper will be more closely examined later, as this is the data we use for our experiments.

3 Approach

3.1 Baseline Approaches

3.1.1 BERT-to-BERT Encoder-Decoder

We used the Huggingface API to build two encoder-decoder models with input defined as a line of pseudocode and output as a line of c++ code [2]. The API allows for the specification of separate encoder and decoder model configurations, and adds cross-attention weights along with a language modeling head for the decoder. The success of BERT has been demonstrated in various downstream tasks, and is widely used for many applications [3]. Using a model trained exclusively on English datasets, on a C++ translation task follows the transfer learning success of neural machine translation models in both zero-shot learning contexts and finetuning tasks, especially in the case where one language is low resource [4][5][6].

For the first model, we used a pretrained, uncased BERT model as our encoder, with default BERT input tokenization. For the decoder, we used a BERT decoder with a language modeling head, with random weight initialization. We did not use a pretrained decoder for this problem, since we wanted to experiment with custom C++ tokenization, as explored in Kulal et al[1]. The authors used a clang parser to tokenize the C++ code, and released their tokenized data for the training, validation and test sets. This allowed us to train a custom C++ tokenizer directly on tokenized training dataset. The model was trained end to end on the SPoC dataset on the training split. This meant that effectively, the encoder was finetuned while the decoder was trained directly on the supervised task.

For the second model, we used a pretrained, uncased BERT model as our encoder and as our decoder, with default BERT input tokenization for both the input and output. We hypothesized that this would allow us to employ the model's learned representations for language structure, allowing us to finetune on pseudocode (English) to code (C++) translation.

3.1.2 BART

We explore the use of a fully pretrained encoder-decoder model based on prior work establishing performance improvements over LSTMs for sequence generation tasks [7]. To do so, we use the Huggingface API [2] to access a BART base model (12-layer, 768-hidden, 16-heads, 139M parameters)

based on BART's documented success on sequence generation and machine translation [8]. As a baseline, we train a model to take one like of pseudocode as input and output the corresponding line of pseudocode. For example, the input `declare integer i` would correctly output `int i;`.

Our use of this model serves dual purposes. First, it evaluates the performance of the Transformer architecture on the task of pseudocode-to-code translation on a line level. We finetune our BART model for the same task Kulal et al perform using an LSTM [1]. This allows us to isolate and quantify the performance impact of the Transformer architecture. Given that work using Transformers for the pseudocode-to-code translation has not, to the best of our knowledge, been reported, this result is meaningful in its own right. Second, it serves as a baseline for modifications we propose.

3.2 Context Awareness

A key challenges facing program-level correctness in pseudocode-to-code translation is that two lines of code that are functionally correct in isolation may be incorrect when paired together. This problem is similar to that of long distance dependencies in general natural language processing work: correct translation of an individual line may rely on context as far back as the beginning of the program. We focus on an approach that makes use of the Transformer's ability to model long range dependencies better than LSTMs [9]. We examine the effect of providing additional context to the model during training and testing.

Initially, we hoped to feed entire input programs to the model. This has a notable limitation: the Transformer architecture places a limit on the limited number of positional embedings available to an input. This prevents an arbitrarily long input length and prevents longer code examples. The limit was exceeded by examples in our dataset and any practical application is likely to do the same. A second consideration was that the compute required to train longer input lengths would have prevented multiple trials. Finally, using the addition of our prefix to translate one pseudocode line to one C++ line allows us to compare our performance with our baseline.

To accomplish this, we format the problem as a general text-to-text problem based on prior success of other models using general text-to-text formatting to translate one line at a time [10]. To translate line l_i of pseudocode into candidate translation t, we use an input format providing a prefix of n lines $p_1, \ldots, p_n$ followed by l_i. $p_1, \ldots, p_n$ are preceded by a `[PRE]` token and concatenated with `[ENDL]` tokens, and l is preceded by a `[PRED]` token. The `[PRE]` and `[PRED]` tokens follow a standard format for text-to-text problems and the `[ENDL]` tokens preserve the meaning of line breaks, which is significant in formats like pseudocode [10]. These tokens are added to the model's vocabulary at finetune time with randomly initialized weights. Based on this input the model predicts t the C++ translation of l. The advantage of this format is that it allows us to compare prefixes of different types and lengths to evaluate the effects of the type of prefix content used. The input format we use is shown for a general prefix in Figure 2.

Before describing our input format approaches, we will note a limitation shared by both. Namely, they rely on the preceding n lines of pseudocode will contain relevant information for translating l_i. This depends on the size of n and the content of $l_{i-n}, \ldots, l_{i-1}$. It often occurs that a variable is declared early on in the function's scope, for example, and as a result is not available to the model for consideration during prediction. The average program length in our data set being about 15 lines, however, convinced us that this would be a meaningful amount of context with respect to our task [1].

3.2.1 Pseudocode Prefix

The first prefix type we consider is n lines of pseudocode. To translate line l_i of the pseudocode into line t_i of C++, we provide $l_{i-n}, \ldots, l_{i-1}$ as $p_1, \ldots, p_n$.

Our train, eval, and test datasets draw on the original SPoC datasets, drawing prefix lines $p_1, \ldots, p_n$ from gold translations $l_{i-n}, \ldots, l_{i-1}$. l_i and g_i are the same as our baseline.

3.2.2 Code Prefix

The second prefix type we consider is n lines of code. To translate line l_i of the code we provide $c_{i-n}, \ldots, c_{i-1}$ as $p_1, \ldots, p_n$. $c_{i-n}, \ldots, c_{i-1}$ are the n lines of C++ code preceding the candidate translation t_i. Though a `;` token is syntactically equivalent to our `[ENDL]` token, we maintain its use for consistency.

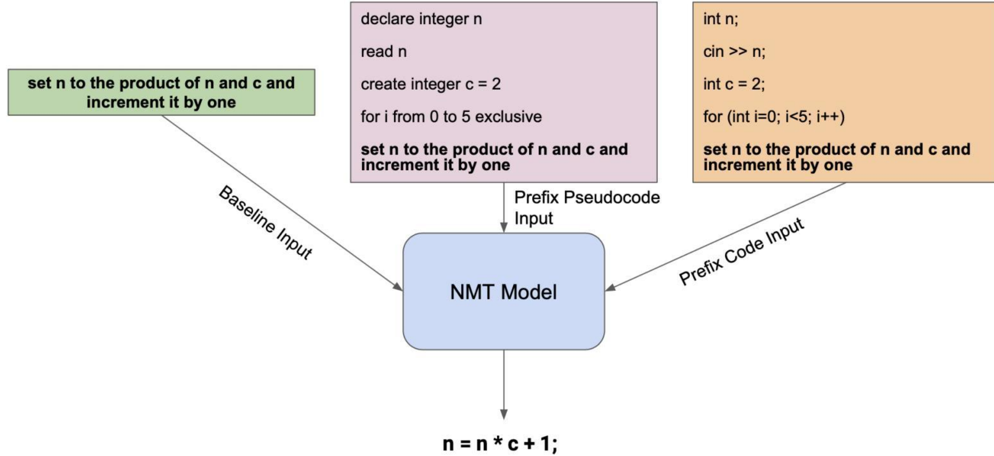


Figure 1: Different input types visualized, with prefix length N=4.

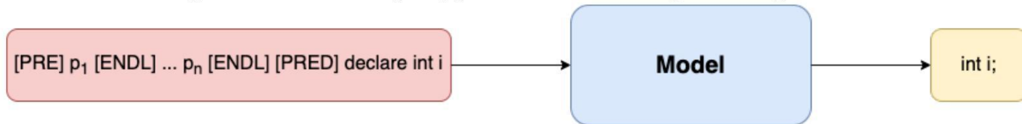


Figure 2: Generic prefix format used to provide context for translation. $p_1, \ldots, p_n$ are lines 1 through n of the prefix.

Our train, eval, and test datasets draw on the original SPoC datasets, drawing prefix lines $p_1, \ldots, p_n$ from gold translations $g_{i-n}, \ldots, g_{i-1}$. l_i and g_i are the same as our baseline. This is equivalent to assuming that during the prediction of t_i, $t_{i-n}, \ldots, t_{i-1}$ have been correctly predicted. This obviously diverges from the realistic case, in which even functionally correct lines may not match the gold translation. We feel this is justified because meaningful information about the effects of providing a code prefix is still retrievable from these trials. Seeing an additional benefit of including code prefixes would point towards confirming the hypothesis that code context is relevant and useful for making better line translations, and in the process mimics a more holistic approach to program translation as opposed to isolated line translations.

4 Experiments

4.1 Data

We are using the SPoC pseudocode-to-C++ code dataset published by Kulal et al[1]. The code portion of the dataset comprises of 18,356 C++ programs, scraped from the solutions to the easiest competitive programming problems on *codeforces.com*. The programs are about 15 lines each on average. The pseudocode was generated by 59 Amazon Mechanical Turk crowdworkers (workers were tested on programming aptitude), and the pseudocode descriptions of programs are on a line-by-line basis, meaning that each line in the pseudocode has a corresponding line of code. Furthermore, the dataset allows for automated checking for functional correctness by providing an average of 39 test cases per program, again scraped from *codeforces.com*. There were two different test splits. Each line is provided with the indentations and the scope brackets removed. This allows the model to learn translations on a line-by-line basis without having to worry about scope issues. There were ∼216,000 lines in the entire dataset. The split was approximately 85-9-6% for training-val-test, and the authors provided two different splits of the data. The first was split on the problem level, to ensure the model had not seen a version of the program solution in the training set. The second split was on the worker level, which meant that some solution to 27% of programs were present in both the training and test sets.

Our tests use the split by worker, as this was the more challenging of the two for an LSTM. For all tests, we preprocess C++ lines using a clang-based tokenizer. For example, the line "`arr.push_back(5);`" would appear as "`arr . push_back ( 5 ) ;`". We further preprocess the dataset such that each

example is trained in accordance with the generic input format we specify (Figure 2) and the specifics outlined in 3.2. If fewer lines are available than the n lines specified by a model's prefix format, we use all previously available.

4.2 Evaluation method

The SPoC dataset includes test cases. Comparing the outputs of the candidates generated by our model with these expected outputs, which evaluates functional correctness, is the basis of our evaluation metric.

Call a pseudocode program p and its gold translation g. Call the prediction set of cardinality K for p_l (top K candidate translations for p_l) $C_l(K)$. We first define the "Line Oracle" test at rank K: Candidate set $C_l(K)$ passes the Line Oracle test if at least one member of $C_l(K)$ may be substituted for g_l without loss of correctness on provided test cases on the program level. This metric is calculated on a line level. We also define the "Program Oracle" test at rank K. A program p passes the "Program Oracle" test at rank K if for each line g_l, when we substitute it with some member of $C_l(K)$ (and keep the rest of g intact), the program passes all test cases. This metric is calculated on a program level, and effectively tests if the Line Oracle at rank K passes for all lines in the program. These metrics, also used by Kulal et al. are better than the BLEU score frequently used for machine translation because, assuming the pseudocode is a valid solution, a semantically correct translation from pseudocode to code will produce a program that passes the provided test cases[1]. Of note, success rate on the Program Oracle metric indicates the number of programs for which success would be theoretically possible if the outputs were used for a search based method with a the top K candidates for each line.

Our goal in translating a line is to maximize the joint probability of the token sequence. Denoting a candidate translation by $(t_1, t_2, ..t_n)$, where t_i is a token, the probability we maximize is $p(t_1, t_2..t_n; \theta)$ where model parameters are denoted as θ. The decomposition of the probability into $\prod_i p(t_i|t_{<i}; \theta)$ allows us to decode token by token. In order to avoid a greedy decoding method, we utilized beam search with beam size of 100 for our test predictions. We then report the 100 candidate translations represented by the beams, in the order of their probabilities assigned by the model.

4.3 Experimental details

4.3.1 BERT Models

The BERT-to-BERT encoder decoder models were trained with stochastic gradient descent on 35,000 lines (approximately 315,000 tokens), which took about 8 hours each on a single GPU on Microsoft Azure. Both models use default training parameters provided by Huggingface were used for both experiments unless otherwise noted, with an initial learning rate of $5 \cdot 10^{-5}$ and an Adam optimizer with β parameters of (0.9,0.999). Furthermore, we used a linear warmup (of 2000 examples) to the initial learning rate so as to avoid early over-fitting.

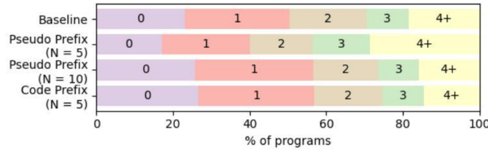
4.3.2 BART Models

For every trial using a BART encoder-decoder, we maintained an identical set of initial conditions and hyperparameters. We recognize that additional hyperparameter tuning may have resulted in more accurate final models, but we chose to forgo this tuning in favor of analyzing the effects of our input formats.

We used the pretrained `bart-base` model (12-layer, 768-hidden, 16-heads, 139M parameters) available as a PretrainedEncoderDecoder from the Huggingface API. We used default training parameters provided by Huggingface were used for both experiments unless otherwise noted, with an initial learning rate of $5 \cdot 10^{-5}$ and an Adam optimizer with β parameters of (0.9,0.999). Each of our trials trained for 6 epochs with a batch size of 40 over 181,990 examples for a maximum of 27300 optimization steps. We evaluated every 3000 steps on our full evaluation set and allowed for early stopping if 3 evaluations end without improvement over the best eval loss. We save a checkpoint with each evaluation, totaling a maximum of 5 checkpoints. Training was preformed using a Microsoft Azure NC12_Promo VM with two Nvidia Tesla K80 GPUs. Finetuning was preformed using a modified script originally based on a Huggingface training example [1].

[1] https://github.com/huggingface/transformers/tree/master/examples/seq2seq

(a) Number of lines in a program where the top candidate is incorrect, i.e. $C_l(1)$ fails



(b) Number of lines in a program where no candidate is correct, i.e. $C_l(100)$ fails.

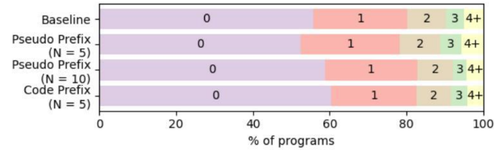
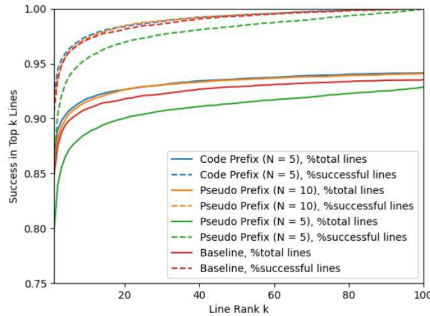
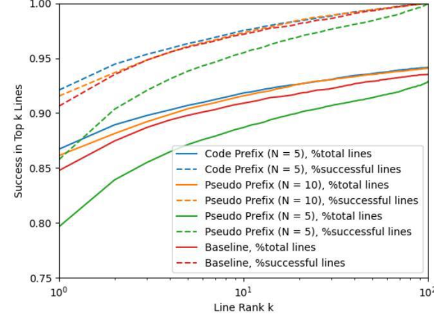


Figure 3: Line Oracle test results per program in percentage of number of programs for the Baseline, Pseudocode Prefix, and Code Prefix fine-tuned BART models with with beam size 100.



(a) Cumulative success in $C_l(k)$ (top k lines)



(b) Cumulative success in $C_l(K)$ (top k lines) on a logarithmic scale

Figure 4: Line Oracle test results demonstrating cumulative success in $C_l(K)$ (top K lines), ratio of all lines and in percentage of lines with a successful candidate in $C_l(100)$, reported for the Baseline, Pseudocode Prefix, and Code Prefix fine-tuned BART models with beam size 100.

Across our experiments, we vary our input prefix type and length. Our baseline is our BART model finetuned with no prefix. Using our pseudocode-prefix and code-prefix input formats, we train with $n = 5$ for both formats and $n = 10$ for prefix pseudocode. The actual training time varied by maximum input length from approximately 8 hours on our baseline with a 96 token maximum input length to approximately 24 hours for $n = 10$ with a maximum input length of 500 tokens. In practice, only the baseline model made use of the early stopping. After finetuning, we select the best model from the available checkpoints for each trial and preform predictions over our test set using beam size 100 and output the top 100 predictions for each line of pseudocode in keeping with the procedure outlined by Kulal et al. [1].

4.4 Results

4.4.1 BERT-to-BERT Encoder-Decoder

Neither of the models were able to produce syntactically correct code translations, and no functional correctness metrics were run on these models. This performance can be attributed to two factors. First, since the cross-attention weights are randomly initialized, it is difficult for the model to learn the cross-linguistic dependencies such as variable names that are persistent across the two languages. Second, the models had $\sim$230M parameters, requiring considerable training on a task in other to generate meaningful results; this was not possible under the time constraints. The model with the untrained decoder had the additional complication that its decoder was not pretrained on any language modeling task, making it significantly more difficult to learn language representations on a small dataset and little training.

Rank=K	Baseline	Pseudo Prefix N = 5	Pseudo Prefix N = 10	Code Prefix N = 5	SPoC*
1	84.8	79.7	86.1	**86.7**	84.0
5	89.8	87.1	90.3	**90.7**	89.1
10	90.9	88.7	91.6	**91.8**	90.0
100	93.5	92.9	94.1	**94.2**	92.0

(a) Line Oracle success (%) at ranks 1, 5, 10, and 100, demonstrating the percentage of all lines where there is a successful candidate in $C_l(K)$ (top k candidates)

Rank=K	Baseline	Pseudo Prefix N = 5	Pseudo Prefix N = 10	Code Prefix N = 5	SPoC*
1	23.1	17.1	25.7	**26.7**	18.2
5	37.8	29.0	40.1	**40.3**	N/A
10	43.6	35.0	45.7	**46.0**	N/A
100	55.8	52.5	58.7	**60.4**	55.2

(b) Program Oracle success (%) at ranks 1, 5, 10, and 100, demonstrating the percentage of all programs where there is a successful candidate in $C_l(K)$ for each line of the program

Table 1: Line and Program Oracle test results for the Baseline, Pseudocode Prefix, and Code Prefix fine-tuned BART models with beam size 100. *SPoC results displayed for comparison

4.5 BART

As can be seen in Table 1(a) our baseline achieves a success rate of 84.8% on our Line Oracle success metric and success rate of 55.8% on our Program Oracle success metric. These are modest improvements over the 84.0% and 55.2% achieved by the LSTM model used in Kulal et al[1].

4.5.1 Pseudocode Prefix

5 Lines We find that finetuning with a pseudocode prefix of length 5 decreases preformance on all metrics relative to our baseline. Rank 1 Line Oracle falls to 79.9%. Our Line Oracle metric is 52.5%. Rank 100 is 92.9%, however, which remains an improvement over SPoC's 92.0% [1]. These results show lower accuracy than we expected, possibly because a prefix of length 5 did not offer enough useful information and instead confused the model's predictions.

10 Lines Providing a prefix of 10 pseudocode lines improves significantly over 5 lines. Rank 1 Line Oracle is 86.1%, rank 1 program oracle is 25.7%. Our rank 100 results for these two metrics are 94.1% and 58.7% respectively. Across all measured metrics (see Table 1), this approach outperforms the SPoC LSTM and our baseline [1]. Additionally, it offers significant improvement over the same approach using a shorter prefix, supporting our hypothesis that providing additional context to aid translation improves performance. Notably, it increases the percentage pf potentially solvable programs in the test set from 55.2% to 58.7% by providing a correct translation in the top 100 candidates for each line in those programs.

4.5.2 Code Prefix

5 Lines Using a prefix consisting of 5 preceding lines of code outperformed our baseline and pseudocode prefix approach of either length. Rank 1 Line Oracle is 86.7%, rank 1 program oracle is 26.7%. Rank 100 results for these two metrics are 94.2% and 60.4% respectively. This was achieved despite having fewer lines of contextual information that our trial using 10 lines of prefix pseudocode. Accounting for the simplifying assumption that preceding output lines are correct, these results support the conclusion that the content of contextual information plays a large role in the model's ability to make use of it for prediction. Furthermore, it indicates that knowledge of preceding code is more useful for predicting the correct translation than preceding pseudocode. More generally, the content of contextual information has a larger affect than the amount. Intuitively, this makes sense because the model has access to the syntax it must match to produce a functional line in addition to the functional goal.

5 Analysis

Qualititive Output Analysis The model output errors often made small and relatively understandable bugs, outputting lines that seem syntactically correct like `if ( a [ q ] == 1 ) b [ count ++ ] = a [ Q - 1 ] ;` . In the correct version, in the correct version, `Q` should be lowercase and `count` should be pre-decremented. Outputs that often produce functionally equivalent code include omissions of brackets in single line if statements and for loops.

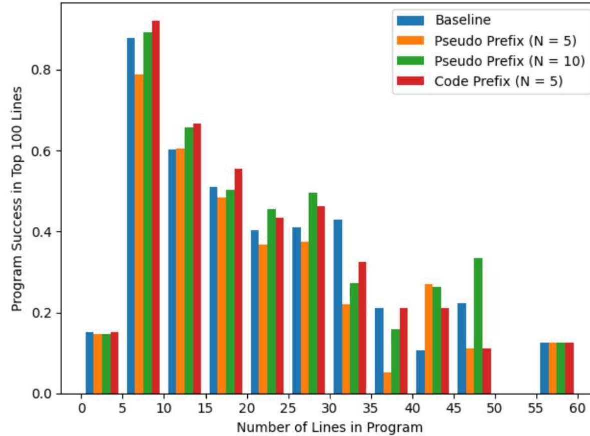


Figure 5: Program Oracle test results demonstrating the ratio of programs within a given length range where there is a successful candidate in $C_l(100)$ (top 100 candidates) for each line of the program, reported for the Baseline, Pseudocode Prefix, and Code Prefix fine-tuned BART models with beam size 100.

Figure 4(b) shows that there could be a linear relationship between $log(K)$ and the success percentage at rank K. This has an intuitive interpretation in the size of the prediction set: There are diminishing returns in the additional benefit of including another prediction in the candidate set, and the returns are inverse proportional to the set size. The additional benefit of going from candidate set size 1 to 10 is approximately equivalent to the added benefit of going from candidate set size 10 to 100. There is a progressively smaller marginal return for additional candidates using more or higher quality prefix information. This indicates the additional information allows the model to more accurately evaluate the relative correctness of the candidates it produces.

Figure 5 shows a relationship between the amount of context provided (in terms of number of prefix lines) and the relative correctness of the different models on different program lengths. The similarity of performance on small program lengths is expected since the small number of available prefix lines prevents context-enhanced models to take full advantage for much of the program. The advantages of context-aware models are largest between in the 10-20 line range, indicating context is generally most helpful when it captures a large portion of the program.

6 Conclusion

The task of pseudocode to code translation introduces new challenges in addition to those presented by natural language translation, and creates new avenues for experimentation. The requirement of syntactic correctness and cross-line dependencies in the translations opens the possibility of experimenting with different tokenizers, along with providing more demanding and robust performance metrics. Following in a similar framework to Kulal et al's work, we were able to match the performance compared to an LSTM on similar metrics[1]. Different input types had varying impacts on performance, with 5 lines of prefix code performing best but 10 lines of pseudocode prefix performing comparably. In particular, while code-prefix input is not feasible from an entire-program generation perspective, its success supports the intuition that cross-line dependencies can be picked up and utilized by a model for more accurate translations. Based on improvements to the number of programs theoretically solveable by search methods (measured by Program Oracle) indicate transformers, especially with prefix based context, can positively contribute to program synthesis. Other avenues for beneficial future work include prediction using previously generated candidates as code prefix lines, trials investigating longer prefixes, and investigations of the effectiveness of pretraining on C++ code for the purposes of code translation.

References

[1] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code. *CoRR*, abs/1906.04908, 2019.

[2] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[4] Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda B. Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's multilingual neural machine translation system: Enabling zero-shot translation. *CoRR*, abs/1611.04558, 2016.

[5] Alexandra Chronopoulou, Dario Stojanovski, and A. Fraser. Reusing a pretrained language model on languages with limited corpora for unsupervised nmt. *ArXiv*, abs/2009.07610, 2020.

[6] Shiyue Zhang, Benjamin Frey, and Mohit Bansal. Chren: Cherokee-english machine translation for endangered language revitalization, 2020.

[7] Sascha Rothe, Shashi Narayan, and Aliaksei Severyn. Leveraging pre-trained checkpoints for sequence generation tasks. *CoRR*, abs/1907.12461, 2019.

[8] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461, 2019.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[10] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.