# SANTA CLARA UNIVERSITY
**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**


Date: June 7, 2022

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

**Colin Rioux**

ENTITLED

## Deep Learning Pseudocode Generation: A Qualitative Analysis

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE COMPUTER SCIENCE AND ENGINEERING

Ihan Hsiao (Jun 7, 2022 14:21 PDT)

Thesis Advisor

David C. Anastasiu
David C. Anastasiu (Jun 7, 2022 14:24 PDT)

Reader

N. Ling
N. Ling (Jun 7, 2022 16:11 PDT)

Department Chair

# Deep Learning Pseudocode Generation: A Qualitative Analysis

by

Colin Rioux

Submitted in partial fulfillment of the requirements
for the degree of
Master of Science Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 7, 2022

# Deep Learning Pseudocode Generation: A Qualitative Analysis

Colin Rioux

Department of Computer Science and Engineering
Santa Clara University
June 7, 2022

ABSTRACT

Pseudocode is a traditional teaching tactic in computer science, yet it is not standardized and programming language dependent. Thus, it can be quite time consuming to write it. With the advancement of AI methodologies in NLP, AI could help address this problem. This work investigates the quality of AI generated pseudocode from source code. Five studies are conducted in this work to measure pseudocode quality, where each study modifies model input to observe accuracy and generalizability. The results show that there is an association between pseudocode quality and training and test set similarity. Furthermore, a sizable and diverse training set and a same language test set is critical for good quality generated pseudocode. Future work can explore language independent embeddings to simplify datasets while maintaining language semantics, if the creation of more applicable datasets is unfeasible.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Sharon Hsiao for her guidance and assistance throughout this process. This work could not have been done without her. I would also like to thank my lovely partner, Lya, who encouraged me to pursue this thesis and pursue my dreams. Her continued support and willingness to help has motivated me the last two years and here's to many more. I would also like to thank Alex Pacifico, Wilson Deng, and Will Heller, who helped me formulate my thoughts when writing my thesis.

# DEDICATION

I dedicate this work to my late friend Shubha Debnath. The last three years have flown by and I miss you so much. Rest easy, my brother.

# Table of Contents


**1 Introduction** **1**
1.1 Motivation . . . 1
1.2 Objective . . . 2
1.3 Research Questions . . . 2

**2 Literature Survey** **3**
2.1 Why Pseudocode? . . . 3
2.1.1 Intermediate Representations . . . 4
2.1.2 Pseudocode . . . 5
2.2 Pseudocode Generation From Source Code . . . 5
2.3 Source Code Generation From Pseudocode . . . 6
2.4 Source Code Generation From Natural Language . . . 6
2.5 Pseudocode in Education . . . 7

**3 Methods** **9**
3.1 DeepPseudo . . . 9
3.2 Datasets . . . 9
3.2.1 SPoC . . . 10
3.2.2 Django . . . 10
3.2.3 StaQC . . . 10
3.3 Evaluation Measures . . . 11
3.3.1 BLEU . . . 11
3.3.2 METEOR . . . 11
3.3.3 ROUGE . . . 11
3.3.4 CIDEr . . . 12
3.3.5 Cosine Similarity . . . 12
3.3.6 Visual Examination . . . 12
3.4 Execution Environment . . . 12

**4 Results** **13**
4.1 DeepPseudo Replication Study . . . 13
4.2 Multi-Line Pseudocode Generation Study Results . . . 14
4.3 Generalized Multi-Line Pseudocode Generation Study Results . . . 16
4.4 Language Specific Multi-Line Pseudocode Generation Study Results . . . 17
4.5 Language Specific Multi-Line Pseudocode Generation via Single Line Processing Study Results . . . 18

**5 Discussion** **19**
5.1 DeepPseudo Replication Study . . . 19
5.2 Multi-Line Pseudocode Generation Study . . . 20
5.3 Generalized Multi-Line Pseudocode Generation Study . . . 20
5.4 Language Specific Multi-Line Pseudocode Generation Study . . . 20
5.5 Language Specific Multi-Line Pseudocode Generation via Single Line Processing Study . . . 21

**6 Future Work** **22**
6.1 Improved Datasets . . . . . . . . . . 22
6.2 Using an Abstract Syntax Tree as the Input Embedding . . . . . . . . . . 22

**7 Conclusion** **23**

# List of Figures

2.1 An example of a natural language summary and source code pair [1]. . . . . 3
2.2 Relationships between natural language phrases and source code snippets [1]. . . . . 3
2.3 Relationships between NLSegments in English and French. . . . . 4
2.4 Role of IRs in language compilation. Source: Adapted from [2]. . . . . 4
2.5 Example pseudocode (left) and soure code (right) pair. Source: Adapted from [3]. . . . . 5

3.1 The training process vs. the experiment process. . . . . 10

# List of Tables


4.1 Generated pseudocode from the DeepPseudo algorithm applied to the SPoC test set [3] . . . . . . . . . 13
4.2 Subset of average NLP comparison scores between the true and generated pseudocode results from Table 4.1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.3 Subset of average cosine similarity scores between the true and generated pseudocode results from Table 4.1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.4 Generated multi-line pseudocode from multi-line C++ source code using the DeepPseudo algorithm . 15
4.5 Subset of average NLP comparison scores between the true and generated pseudocode results from Table 4.4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.6 Subset of average cosine similarity scores between the true and generated pseudocode results from Table 4.4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.7 Generated multi-line pseudocode from multi-line Python source code using the DeepPseudo algorithm trained on C++ source code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.8 Generated multi-line pseudocode from multi-line Python source code using the DeepPseudo algorithm trained on Python source code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.9 Generated multi-line pseudocode from decomposed multi-line Python source code using the DeepPseudo algorithm trained on Python source code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

# Chapter 1

# Introduction

If you enter ”learn computer science” into a search engine, the amount of results is astounding. The materials range from interactive websites, to pure documentation, books, tutorials, and videos. Despite the abundance of computer science resources, learning how to or teaching to program remains difficult. Among these resources is a common teaching tactic: pseudocode, whose goal is to help teach algorithmic thinking and code structure without overwhelming the learner with abstract syntax [4, 5].

Pseudocode is diverse: there are general guidelines on how to write it, but there are many variants of it [6]. The most common of these forms is its textual one, which often uses a hybrid of English language statements and the target programming language’s syntax [6, 7]. Pseudocode can be written formally or informally, as long as it upholds its goals of improving a student’s syntax comprehension and idea organization [8, 9].

Block-based programming tools such as Scratch[1] and Alice[2] have become prevalent in computer science education, both of whom are pseudocode-like programming environments [10, 11, 12, 13]. An advantage of these tools are their built-in visualization features, which enable students to see the results of their code in real-time [14]. However, there is some concern that these tools make learning traditional programming more complex by being a bit too different from traditional pseudocode [12, 15]. Even in an age of overwhelming pseudocode adoption, computer science students still struggle to learn how to code.

## 1.1 Motivation

An inexpensive and high quality generator for pseudocode can alleviate the time required to generate it by hand. Artificial Intelligence (AI) has proven itself as a valuable strategy for natural language processing (NLP), especially in language translation [16]. Effective AI models could address the automatic pseudocode generation problem.

[1] https://scratch.mit.edu
[2] https://www.alice.org

## 1.2 Objective

This work aims to research the adequacy of a particular AI method called DeepPseudo to automatically generate pseudocode [17]. A replication study and four extension studies are conducted to verify the accuracy of such a model. Visual examination and statistical analysis are the techniques used to measure accuracy and suggest improvement.

## 1.3 Research Questions

The studies conducted in this work aim to answer the following research questions (RQs). Answering such questions should evaluate the efficacy of the DeepPseudo model while considering AI as a solution to automatic pseudocode generation.

- RQ1: What is the semantic quality of machine generated pseudocode on sentence and document levels?
- RQ2: Can DeepPseudo be generalized to other programming languages?

The rest of this work is structured as follows. Chapter 2 will review natural language semantics, pseudocode, research in pseudocode generation, and the effects of pseudocode in education. Chapter 3 will delve into the DeepPseudo model, datasets used, how evaluation will be conducted, and the evaluation environment. Chapter 4 will examine the results from each study and Chapter 5 will try to explain the meaning of these results as they pertain to my research questions. Finally, Chapters 6 and 7 will suggest next steps and make overall conclusions.

# Chapter 2

# Literature Survey

## 2.1 Why Pseudocode?

To best understand the role pseudocode plays in education, we must understand why the source code depicted in Figure 2.1 corresponds to its summary. At a minimum, these four concepts must be understood: Python syntax, objects, files, and CSV file properties. Additionally, students must also identify a relationship between the natural language and source code.

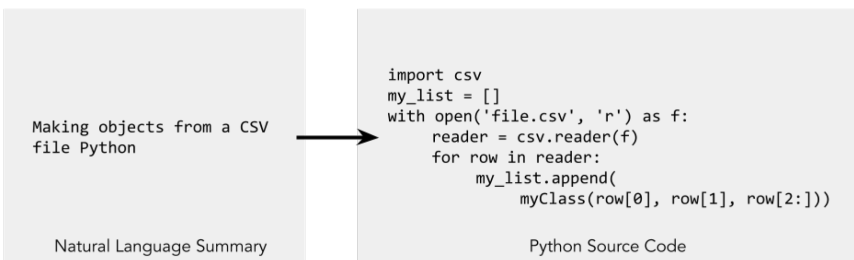


Figure 2.1: An example of a natural language summary and source code pair [1].

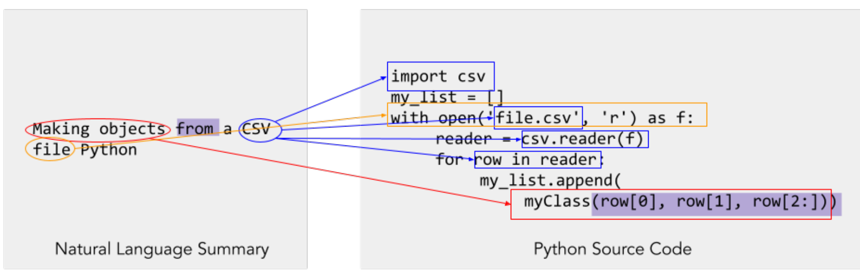


Figure 2.2: Relationships between natural language phrases and source code snippets [1].

Let's define a natural language segment (NLSegment) as a sequence of one or more words. According to Figure 2.2, an NLSegment can correspond to multiple parts of source code. In the case of the NLSegment "from", NLSegments can also operate as a link between other NLSegments. For example, the word "from" implies that a CSV

file must be used to ”make [an] object”. Unlike translating between spoken languages (see Figure 2.3), there is no one-to-one relationship to be made between NLSegments and source code.

Figure 2.3: Relationships between NLSegments in English and French.

### 2.1.1 Intermediate Representations

According to Chow, an intermediate representation (IR) is ”any data structure that can represent the program without loss of information so that its execution can be conducted accurately” [2]. In other words, IRs are used as an intermediary step to reliably convert from the source language to the target language (see Figure 2.4).

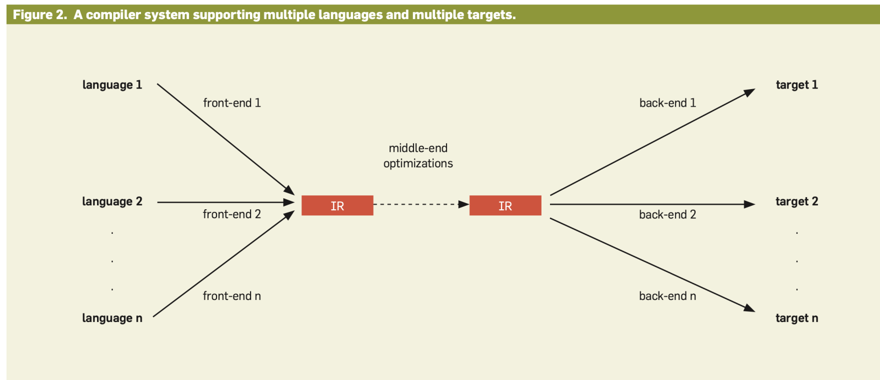


Figure 2.4: Role of IRs in language compilation. Source: Adapted from [2].

Such a concept can be adapted to the conversion of natural language into source code, as there can be infinitely different summaries that can correspond to the same piece of source code. This is likewise the case for source codes as well, since there are many different programming languages that a program can be written in.

Thus, it is only natural to consider an intermediary relationship between natural language and source code, that is flexible enough to manage different summaries and programming languages. An IR, in theory, would address the lack of a one-to-one relationship between natural language and source code, while preserving the goal of the target source code.

### 2.1.2 Pseudocode

Pseudocode is the typical IR between natural language and source code. It is seen whether you are a beginner computer scientist or an adept one, and it has become somewhat of a tradition [18]. Pseudocode seems like an obvious starter IR, as it has natural language conventions and the structure of a target source code [6, 7].

| $i$ | $x_i$ | $y_i$ |
|---|---|---|
| 1 | in function main | int main() { |
| 2 | let n be integer | int n; |
| 3 | read n | cin >> n; |
| 4 | let A be vector of integers | vector<int> A; |
| 5 | set size of A = n | A.resize(n); |
| 6 | read n elements into A | for(int i = 0; i < A.size(); i++) cin >> A[i]; |
| 7 | for all elements in A | for(int i = 0; i < A.size(); i++) { |
| 8 | set min_i to i | int min_i = i; |
| 9 | for j = i + 1 to size of A exclusive | for(int j = i+1; j < A.size(); j++) { |
| 10 | set min_i to j if A[min_i] > A[j] | if(A[min_i] > A[j]) { min_i = j; } |
| 11 | swap A[i], A[min_i] | swap(A[i], A[min_i]); |
| 12 | print all elements of A | for(int i=0; i<A.size(); i++) cout<<A[i]<<" "; |
| | | } |

Figure 2.5: Example pseudocode (left) and soure code (right) pair. Source: Adapted from [3].

Wells and Kurtz explain that traditional pseudocode is "a subset of many target languages" of the same paradigm [19]. In other words, traditional pseudocode references common features found across all of its target languages. This suggests two things. First, pseudocode is not a standardized convention. The common features found across its targets will change overtime. Second, pseudocode is not general enough to target all languages across different paradigms.

## 2.2 Pseudocode Generation From Source Code

Pseudocode generators address the massive developer overhead to write pseudocode by hand. Generation is especially difficult given pseudocode has no standard. This section reviews two source code to pseudocode generators.

Statistical Machine Translation (SMT) is a common technique used to generate pseudocode from source code. The problem with such a technique is that it is dependent upon its training/test datasets. The work proposed by Alhefdhi et al. uses Neural Machine Translation (NMT) instead to automatically generate pseudo code from source code [20]. Their approach leverages the typical encoder/decoder model seen in natural language processing (NLP), as an LSTM-based encoder and decoder is used to parse input code into a vector into pseudo-code. The authors used Pseudogen [21] applied on the Django code-base for their experiments [22]. Compared to Tree-to-String SMT (T2SMT), a state-of-the-art SMT approach, this work performs slightly better.

Yang et al. take a different approach by using a convolutional neural network (CNN) for code feature extraction and a transformer encoder/decoder for sequence to sequence translation [17]. Together, their DeepPseudo model learns local and global context which better generalizes the resulting pseudo-code. Yang et al. show that the self-attention mechanism used by the transformer architecture has added benefits and advanced attention models may yield increased accuracy in the future.

## 2.3 Source Code Generation From Pseudocode

Source code generation is a difficult task because the algorithm used must know the syntax and semantics of the target programming language. Since pseudocode has syntax similarities to source code, pseudocode seems like an obvious starting point. This section reviews an AI model and dataset that addresses the source code generation problem using pseudocode.

Kulal et al. introduce the Pseudocode-to-Code (SPoC) dataset, a dataset consisting of 18,356 C++ programs with their corresponding pseudocode representation [3]. They also introduce two error localization methods to improve their candidate code selection: "multiclass classification" and "prefix-based pruning" [3]. Multiclass classification is a neural network approach which predicts the likelihood of single errors within the candidate code. Prefix-based pruning, on the other hand, leverages the power of the compiler to detect any incorrect code prefixes. After applying their work on their new dataset, the authors concluded that their search based approach improves program "synthesis success rate" from "25.6% to 44.7%" [3].

## 2.4 Source Code Generation From Natural Language

Source code generators from pseudocode requires the pseudocode to already exist. Often, it is much easier to write natural language summaries than write pseudocode. This section reviews four algorithms that generate or capture source code from a natural language equivalent.

Mandal and Naskar propose an NLP technique which automatically generates code equivalent to mathematical word problem (MWP) texts [23]. Their technique leverages the convenience of object oriented programming (OOP), as MWP texts and their corresponding sentences can be represented as entities with state and OOP uses the concept of objects to represent entities and their states. Similar to Lu and Hsiao, the authors map MWPs to concepts, except concepts take the form of entities and their corresponding states. To do this, they use the Semantic Role Labelling (SRL) technique. The authors identified the relationship between verbs and equation attributes such as '=' and '+' and used this relationship to generate code equivalences for the entity and state objects belonging to the sentences in the MWP texts. The authors formed a custom dataset composed of 189 MWP texts and used the output of the executed code to verify accuracy.

Yen et al. propose a system that they call SAGE which can determine ambiguity in RFC protocol documents and, after resolution of these ambiguities, can generate code implementation for the protocol [24]. The authors leverage RFC protocol documents that employ a standard document-specific syntax and semantic structure. Despite this advantage, the authors find that these protocol documents are still highly ambiguous. To determine ambiguities from their preprocessed text, the authors use Combinatory Categorial Grammar (CCG) to convert text into logical expressions. If there is 0 or more than 1 logical expression generated for each sentence of text, then the text is considered to be

ambiguous and must be fixed. After logical expression generation, this form is converted into code. The target OS, hardware, and/or software is required in order to successfully generate the code. To verify their system, Yen et al. ran the resulting code to evaluate correctness.

Nguyen et al. developed a tool that converts a short natural language description of a program into a set of API calls [25]. Their three stage model consists of a translator, an expansion model, and an ordering model. The translation stage consists of mapping individual words to their corresponding API elements. To improve the translation stage, a contextual expansion model is applied based on pre-existing and correct mappings. After translation and expansion, a graph-based ordering model called GraLan is used to estimate the likelihood of a sentence occurring in the target language.

Yao et al. propose a deep, hierarchical neural network (HNN) used to capture natural language and source code pairs from Stack Overflow [1]. This model, called BiV-HNN, intelligently analyzes Stack Overflow questions and answers, by identifying complete solutions to well structured questions in Python or SQL. From their work, the authors introduce the StaQC dataset, a natural language to source code dataset which is of higher quality than previous heuristics. It is important to note that the matched source code is evaluated based on quality, not accuracy.

## 2.5 Pseudocode in Education

For automatic pseudocode generation to be considered important, the role pseudocode has in education should be understood. This section reviews multiple case studies, difficulties, and use cases of pseudocode in education.

Copus and Copus conducted a case study to determine a positive correlation, if any, between the quality of pseudocode and source code written by beginner students [18]. Their work found that ”pseudocode quality and ultimate answer quality are indeed corollary” [18]. In other words, a good grasp of pseudocode can result in better source code. However, they could not determine if pseudocode quality came from source code quality, or vice versa.

Olsen proposed a teaching approach where students first learn how to write pseudocode from problem statements and then they learn how to write source code from pseudocode [26]. They evaluated their approach by designing an introductory computer science course built around it. They found that the plethora of pseudocode practice improved the students’ pseudocode quality. Furthermore, their data indicated that students understood computational theory better.

Bosse et al. conducted a case study to identify the difficulties students face while programming by surveying 16 instructors and 110 students [27]. They found that there were mixed opinions on the success of pseudocode in the classroom. In fact, one teacher felt that pseudocode made teaching more difficult. Another instructor said that it ”helps with concepts” only [27].

Malik et al. proposed a teaching approach that focuses on programming knowledge and algorithmic thinking

equally [4]. Their approach asks students to solve problems using pseudocode and then convert the pseudocode into source code. This approach is very similar to Olsen's [26]. This approach was tested via an introductory programming course. They found that students were able to grasp execution flow and programming concepts better.

# Chapter 3

# Methods

We present five studies: one replication study and four extension studies. The replication study replicated the RQ1 experiment of Yang et al. [17]. The purpose of this study is to gather the results of the DeepPseudo[1] model when trained and tested in our execution environment. The four extension studies either train the DeepPseudo model on new data, test it on new data, or both. These studies aim to push the DeepPseudo model to its limits and identify any difficulties it had with the pseudocode generation process.

Each study consisted of three processes, a training process, an experiment one, and an evaluation process. The DeepPseudo model was trained on source code and pseudocode pairs and then tested upon a dataset that either does or does not belong to the same training set (see Chapter 4). The evaluation process consisted of visual examination and statistical analysis. The difference between the training process and the experiment process is depicted in Figure 3.1.

## 3.1 DeepPseudo

DeepPseudo performs sequence to sequence translation using a transformer-based model [17]. Transformers are particularly successful in NLP due to their inherent sequential nature and their ability to identify semantic relationships via self-attention [28]. DeepPseudo was selected over other pseudocode generators because it considers a fundamental part of natural language as we know it: words often depend on the words around it. The same goes for source code. A line of code is considered a machine instruction, which requires a specific sequence of keywords. In addition, DeepPseudo out performed other models solving the same pseudocode generation problem [17].

## 3.2 Datasets

Three datasets were used in this work: the SPoC, Django, and StaQC datasets [3, 22, 1]. Yang et al. used variants of the SPoC and Django datasets in their work [17]. ImageNet, one of the largest training repositories for images, contains 1,281,167 images [29]. As seen in this section, the datasets used in this study are a fraction of the size.

[1] https://github.com/NTDXYG/DeepPseudo

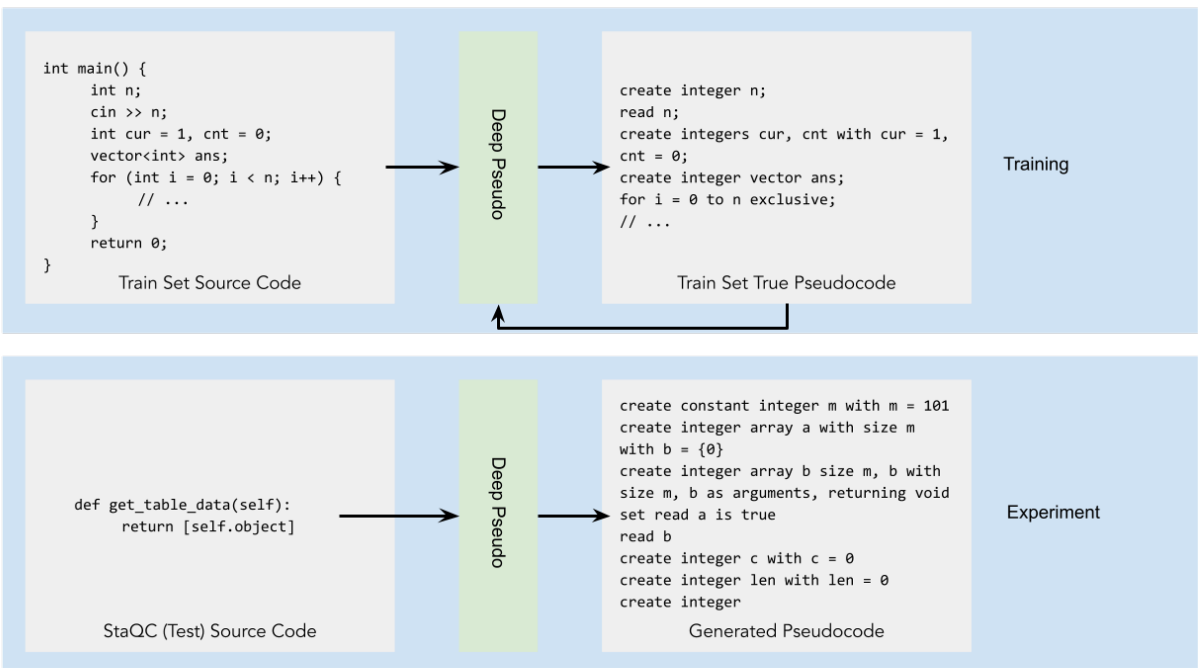


Figure 3.1: The training process vs. the experiment process.

### 3.2.1 SPoC

The SPoC dataset[2] consists of 18,356 C++ source code and pseudocode program pairs [3]. However, the version of SPoC used in the replication study consists of 123,341 single C++ line source code and pseudocode pairs [17].

### 3.2.2 Django

The Django dataset[3] consists of 18,805 Python source code and pseudocode program pairs [22]. Similar to the SPoC dataset, the version of Django used by Yang et al. and this work consists of 15,006 single Python line source code and pseudocode pairs [17].

### 3.2.3 StaQC

The StackOverflow Question Code Dataset (StaQC)[4] is a statistically mined dataset consisting of StackOverflow[5] problem statement and source code pairs [1]. Unlike the other two datasets, this dataset does not include any pseudocode. The purpose of this dataset is to diversify the results seen in different studies by giving the DeepPseudo algorithm source code it has not seen before. This dataset consists of 148,000 Python source code problem and problem statement pairs [1]. It is important to note, however, that what is considered source code in this dataset could be

[2]https://sumith1896.github.io/spoc/
[3]https://github.com/NTDXYG/DeepPseudo/tree/main/data/django
[4]https://github.com/LittleYUYU/StackOverflow-Question-Code-Dataset
[5]https://stackoverflow.com

pseudocode as the miner did not distinguish between the two.

## 3.3 Evaluation Measures

Five statistical and one visual metric are used to evaluate generated pseudocode in this work. The DeepPseudo algorithm uses a tool created by Sharma et al. to automatically compute these statistical metrics [17, 30]. In all, the metrics used are BLEU, METEOR, ROUGE, CIDEr, cosine similarity, and visual comparison.

### 3.3.1 BLEU

BiLingual Evaluation Understudy (BLEU) addresses the cost of human evaluation of machine translation by defining a cost effective evaluation metric [31]. BLEU evaluates the similarity between a machine translation and a human one. BLEU scores range from 0 to 1, where a score of 1 means an identical translation and a score of 0 means nothing alike. BLEU uses n-grams to measure precision. In other words, BLEU counts the occurrences of all sequences of n words and divides it by the total amount of words. Like the work of Yang et al., this work considers four BLEU scores: `BLEU_1`, `BLEU_2`, `BLEU_3`, and `BLEU_4` [17]. Each suffix of the score corresponds to which n-gram is being considered. Therefore, `BLEU_1` is the unigram precision score whereas `BLEU_4` is the 4-gram precision score [31]. Ultimately, the `BLEU_4` score tells us the most about how similar the generated translation is to the human one.

### 3.3.2 METEOR

METEOR is similar to BLEU in that it addresses automatic machine translation evaluation while using unigrams [32]. Fundamentally, however, METEOR addresses several limitations of the BLEU metric: poor n-gram matching, influence of higher order n-grams on lower order ones, and geometric averaging. Bannerjee and Lavie show that METEOR outperforms BLEU on the same trusted test set though both METEOR and BLEU can provide useful insight on the accuracy of machine translation.

### 3.3.3 ROUGE

Recall-Oriented Understudy for Gisting Evaluation (ROUGE) takes inspiration from BLEU in that it uses n-gram precision to help evaluate summaries [33]. Instead of single sentence evaluation, ROUGE is a metric used to evaluate machine generated summaries with ideal human summaries. Yang et al. specifically use the `BLEU-L` metric, which is an F-score based on the longest common subsequence (LCS) found in the summary [17, 33]. Since pseudocode can be considered a generated summary, ROUGE seems like an applicable metric.

### 3.3.4 CIDEr

Consensus-based Image Description Evaluation (CIDEr) is unlike the other metrics described above, as it is typically used in computer vision [34]. Despite this, CIDEr is still a metric which compares single sentences to a majority consensus. This majority consensus may consist of one or more sentences that are considered the true equivalent of a translation. CIDEr encodes sentences by computing the term frequency, inverse document frequency (TF-IDF) for each n-gram within the sentence. CIDEr is then computed using average cosine similarity and multiplied by 10. As such, the CIDEr metric is shown to be more useful when comparing a machine translation to a ground truth set of sentences, though the metric may hold little meaning when doing a comparison with a single true sentence.

### 3.3.5 Cosine Similarity

Cosine similarity is a common metric used in machine learning (ML). In NLP, cosine similarity converts sentences into vector-space, and takes an angular measure to determine the relationship between two sentence vectors. Such a metric is used to identify the similarity across sentences, but it is completely dependent upon how sentences are converted into vectors. Three variants of cosine similarity are used: Skip-Thought, Embedded Average, and Vector Extrema [35, 30, 36]. Each variant uses their own embedding technique. Skip-Thought's embedding is generated using a neural network and it is a very good measure to determine semantic relatedness between sentences [35, 30]. The Embedded Average measure uses a sentence embedding that is an average of all word embeddings within the sentence [30]. Vector Extrema uses extreme values found in sentences as a defining characteristic of the embedding [36, 30]. Vector Extrema gives a different similarity perspective than the other two cosine similarity metrics.

### 3.3.6 Visual Examination

The StaQC dataset does not contain any pseudocode, thus, visual examination is a necessary evaluation technique for this work [1]. The addition of pseudocode to such a dataset could be proven useful in future work, but it is out of the scope of this research.

## 3.4 Execution Environment

Experiments were conducted on SCU's Wiegand Advanced Visualization Environment (WAVE) High Performance Computer (HPC), a state-of-the-art cluster of high performant servers [37]. These experiments were run on the nodes equipped with two GPUs and signficant amount of memory. These nodes enabled quick training and testing of various DeepPseudo models, as both the training sets and generalization set are very dense.

# Chapter 4

# Results

## 4.1 DeepPseudo Replication Study

Conducting a replication study of the RQ1 experiment of Yang et al. was a necessary step before any of our own experiments [17]. Such a replication study ensures all results presented are computed on the same execution environment. The original experiment trained the DeepPseudo model using single line C++ source code and pseudocode pairs from the SPoC dataset [17, 3]. The resulting DeepPseudo model was then evaluated using the SPoC test set. The results from our replication study are presented in Table 4.1.

Table 4.1: Generated pseudocode from the DeepPseudo algorithm applied to the SPoC test set [3]

<table>
<tr><th>ID</th><th>Source Code</th><th>Generated Pseudocode</th><th>True Pseudocode</th></tr>
<tr><td>262</td><td>for (k = 0;; k++)</td><td>for k = 0 to infinity</td><td>increment k from 0 in a loop</td></tr>
<tr><td>303</td><td>p += p1 * (l[i] - r[i - 1]);</td><td>increment p by p1 * (l[i] - r[i - 1])</td><td>increase p by p1 * (l[i] - r[i - 1])</td></tr>
<tr><td>346</td><td>if (n - 5 * (pow(2, m + 1) - 1) < 0) {</td><td>if n - 5 * pow(2, m + 1) - 1) < 0</td><td>if (n - 5 * (pow(2, m+1) - 1)) is negative</td></tr>
<tr><td>1267</td><td>return dp[total][cnt] = max(d, max(e, f));</td><td>return the value of ans1 is equal to maximum of ( npr )</td><td>return dp[total][cnt] = max of d and max(e, f)</td></tr>
<tr><td>3562</td><td>bool digits[10]0</td><td>create boolean array vis with size 200</td><td>create an array of booleans called digits filled with 0</td></tr>
<tr><td>5445</td><td>if (n <= 1 || n > 3 && (n % 2 == 0 || n % 3 == 0)) return 0;</td><td>if n <= 1 or n > 3 and n % 3 = 0 or n % 3 = 0 return 0</td><td>if n is less than or equal to 1 or n is greater than 3 and ( n % 2 is 0 or n % 3 is 0 ), return 0 from function</td></tr>
</table>

Based on the results (Table 4.1), the verbiage used in the generated pseudocode is different from the true pseudocode. Additionally, the DeepPseudo algorithm established different semantic relationships than the human author of the true pseudocode. Notably, the generated pseudocode for ID 262 of Table 4.1 correctly identifies that k is bounded

by infinity, but it does not specify that k increments. Furthermore, ID 303 shows that the algorithm defined ”+=” as ”increment” whereas the human author defined ”+=” as ”increase”. Despite these wording differences, both pseudocodes are fairly good representations of the source code.

ID 342 and ID 5445 paint a different picture. There is no summary for mathematical expressions. Instead, expressions are unchanged, with the only exception being `&&` and `||`, which the algorithm identifies as equivalents to ”and” and ”or” respectively.

IDs 1267 and 3562 show that the algorithm fails to attach any syntactical meaning to variable names. ID 3562 shows that the array is named ”digits” but the generated pseudocode calls the array ”vis”. Furthermore, the return type in ID 1267 is incorrect, as the source code returns a numerical value, but the pseudocode returns a boolean. Finally, the ID 1267 shows that functions and their parameters have no meaning to the model.

To validate the semantic similarity between true and generated pseudocode, the metrics presented in Tables 4.2 and 4.3 were computed.

Table 4.2: Subset of average NLP comparison scores between the true and generated pseudocode results from Table 4.1

| BLEU_1 | BLEU_2 | BLEU_3 | BLEU_4 | METEOR | ROUGE-L | CIDEr |
|---|---|---|---|---|---|---|
| 0.570740 | 0.470071 | 0.401681 | 0.348377 | 0.360662 | 0.556015 | 3.258573 |

Table 4.3: Subset of average cosine similarity scores between the true and generated pseudocode results from Table 4.1

| Skip Thoughts | Embedding Average | Vector Extrema |
|---|---|---|
| 0.685338 | 0.835205 | 0.730490 |

The BLEU, METEOR, and ROUGE scores ranging from 0.348 to 0.570 (Table 4.2) indicate a low to medium level of similarity between the generated pseudocode and the true pseudocode. The CIDEr score also indicates a low level of general cosine similarity, but this is likely due to the small common term frequency between both pseudocodes. The other three cosine similarity scores ranging from 0.685 to 0.835 Table 4.3, however, indicates a medium to high level of semantic similarity. The BLEU, METEOR, and ROUGE scores (Table 4.2) show how n-grams overlap across the pseudocodes, whereas the three cosine similarity measures indicate how similar the meanings of both pseudocodes are [30].

## 4.2 Multi-Line Pseudocode Generation Study Results

Single line source code to pseudocode translation is fairly effective, as seen in the replication study, but computer programs typically consist of multiple lines of code. If an AI model were to process an entire multi-line program at once, it could capture a higher level of the program semantics as well as generate a general and succinct pseudocode. Therefore, in this section, we adopt the DeepPseudo AI algorithm to experiment with multi-line pseudocode generation [17].

Using other information provided in the SPoC dataset, the single lines of source code are combined into multi-line programs [3]. The same is done to the corresponding pseudocode. Newlines and tabs are replaced by their corresponding escape sequences, `\n` and `\t`. This modified version of the SPoC dataset is used for both training and evaluation. The evaluation results are presented in Table 4.4.

Table 4.4: Generated multi-line pseudocode from multi-line C++ source code using the DeepPseudo algorithm

| ID | Source Code | Generated Pseudocode | True Pseudocode |
|---|---|---|---|
| 225 | `int n, i, a[1111111],`<br>`ans = 0, d[1111111];`<br>`int main() {`<br>`cin >> n;`<br>`for (i = 0; i < n;`<br>`i++) cin >> a[i];`<br>`d[1] = 1;`<br>`for (i = 2; i < n;`<br>`i++) d[i] = d[i / 2] *`<br>`2;`<br>`for (i = 0; i + 1 < n;`<br>`i++) {`<br>`a[i + d[n - i - 1]] +=`<br>`a[i];`<br>`ans += a[i];`<br>`cout << ans << endl;`<br>`}`<br>`return 0;`<br>`}` | `create integers n, i,`<br>`j`<br>`create integers ans,`<br>`n, i, ans with ans = 0`<br>`read n`<br>`for i = 0 to n`<br>`exclusive, read arr[i]`<br>`set ans to 1`<br>`for i = 2 to n`<br>`exclusive`<br>`set d[i] to l[i] / 2]`<br>`* 2`<br>`for i = 0 to i + 1 is`<br>`less than n` | `n, i, a, ans, d =`<br>`integers with ans = 0,`<br>`a, d = integer array`<br>`of size 1111111`<br>`read n`<br>`for i = 0 to n`<br>`exclusive, read a[i]`<br>`d[1] = 1`<br>`for i = 2 to n`<br>`exclusive, d[i] = d[i`<br>`/ 2] * 2`<br>`for i = 0 to i + 1 < n`<br>`a[i + d[n - i - 1]] =`<br>`a[i + d[n - i - 1]] +`<br>`a[i]`<br>`ans = ans + a[i]`<br>`print ans` |
| 817 | `int main() {`<br>`int n, b, p;`<br>`cin >> n >> b >> p;`<br>`cout << (n - 1) * (2 *`<br>`b + 1) << "" "" << n *`<br>`p << "\n";`<br>`return 0;`<br>`}` | `create integers n, b,`<br>`p`<br>`read n read b read p`<br>`print ( n - 1 ) * ( (2`<br>`* ( n - 1 ) * b + 1 )`<br>`print \n` | `create int n, b, p`<br>`read n, b, p`<br>`print (n - 1) * (2 * b`<br>`+ 1), " ", n*p and a`<br>`newline` |

There are a few issues discovered and summarized in Table 4.4. First, the generated pseudocode appears to exhibit notable disparate length compared to the true pseudocode. This seems to be because the DeepPseudo algorithm skips some lines of source code. For ID 225, the bodies of each multi-line for loop are ignored. A slight difference is seen in ID 817, however, as the second to last print statement ignores everything after the mathematical equation. Another noticeable issue is that the algorithm still struggles to define arrays correctly in pseudocode. In fact, ID 225 shows that arrays were skipped over entirely, despite being used later in the code.

The previous results show that the generated pseudocode is a weak translation of the original source code. The semantic similarity metrics presented in Tables 4.5 and 4.6 may suggest otherwise.

The BLEU, METEOR, and ROUGE scores ranging from 0.109 to 0.349 (Table 4.5) and the CIDEr score of 0.498 indicate a very low similarity between the generated pseudocode and the true pseudocode. In contrast, the cosine

Table 4.5: Subset of average NLP comparison scores between the true and generated pseudocode results from Table 4.4

| BLEU_1 | BLEU_2 | BLEU_3 | BLEU_4 | METEOR | ROUGE-L | CIDEr |
|---|---|---|---|---|---|---|
| 0.233310 | 0.172176 | 0.135103 | 0.109659 | 0.182613 | 0.349979 | 0.498249 |

Table 4.6: Subset of average cosine similarity scores between the true and generated pseudocode results from Table 4.4

| Skip Thoughts | Embedding Average | Vector Extrema |
|---|---|---|
| 0.672713 | 0.899808 | 0.694537 |

similarity results (Table 4.6) show a medium to high level of semantic similarity. These results suggest that the generated pseudocode is somewhat similar in meaning to the true pseudocode, but the words and phrases used are not as similar.

## 4.3 Generalized Multi-Line Pseudocode Generation Study Results

Often AI models are evaluated using a test set which originates from the same source as the training set. This is not practical, as a high quality model should be able to generate pseudocode from any source code. In this section, I experiment with DeepPseudo's generalizability by examining the quality of the generated pseudocode with a different training programming language for input. The StaQC dataset, a dataset consisting of multi-line Python programs, is selected for this study [1]. The evaluation results are presented in Table 4.7.

Table 4.7: Generated multi-line pseudocode from multi-line Python source code using the DeepPseudo algorithm trained on C++ source code

| ID | Source Code | Generated Pseudocode |
|---|---|---|
| 750 | from Tkinter import *<br>from Tkinter import Tk<br>nroot = Tk()<br>usernameVal = StringVar()<br>class myclass():<br>def __init__(self):<br>self.main()<br>def submitForm(self,*args):<br>try:<br>print("submitForm pressed")<br>print("name is %s" %<br>self.usernameVal.get())<br>self.usernameVal.set("")<br>except ValueError:<br>pass<br>... | declare void function check with<br>char pointer found with char<br>pointer found to pointer found<br>to 0<br>if not 0<br>if not 0<br>... |
| 756 | pd.options.display.float_format =<br>'${:,.2f}'.format | set found to 1<br>set output to 1<br>set output to output |

Unfortunately, the results demonstrated in Table 4.7 are observed as counterintuitive. The number of generated pseudocode lines in ID 750 likely means that source code lines are being ignored by the model. In addition, the func-

tion "submitForm" seems to have been partially identified, as the first line of the generated pseudocode references a function with a char pointer passed as a parameter. However, the model still fails to syntactically understand the source code by assuming the function is named "check" not "submitForm". In ID 756, the algorithm correctly identifies a relationship between "=" and "set", but the generated pseudocode makes no sense whatsover. Such results show that DeepPseudo cannot generalize well when the language of the training and test sets do not match.

## 4.4 Language Specific Multi-Line Pseudocode Generation Study Results

Pseudocode is a hybrid of natural language and the syntax of its corresponding source code [6]. Therefore, a pseudocode generator must know the appropriate syntactical structure of the original source code. In this section, we train the DeepPseudo algorithm on the Django dataset and test it on the StaQC dataset [22, 1]. The Django dataset consists of single line Python source code and pseudocode pairs [22]. We choose this dataset as it should help the model learn Python syntax. We hope that a different dataset should improve the model's accuracy when evaluated on the StaQC dataset. The evaluation results are presented in Table 4.8. We observe meaningless results in Table 4.8. Not only are

Table 4.8: Generated multi-line pseudocode from multi-line Python source code using the DeepPseudo algorithm trained on Python source code

| ID | Source Code | Generated Pseudocode |
|---|---|---|
| 750 | from Tkinter import *<br>from Tkinter import Tk<br>nroot = Tk()<br>usernameVal = StringVar()<br>class myclass():<br>def __init__(self):<br>self.main()<br>def submitForm(self,*args):<br>try:<br>print("submitForm pressed")<br>print("name is %s" %<br>self.usernameVal.get())<br>self.usernameVal.set("")<br>except ValueError:<br>pass<br>... | substitute<br>settings.cache_middleware_key_prefix<br>for prefix_norm , |
| 756 | pd.options.display.float_format =<br>'${:,.2f}'.format | substitute decimals for op . |

multiple lines of source code ignored, there is minimal Python syntax shown in the generated pseudocode.

## 4.5 Language Specific Multi-Line Pseudocode Generation via Single Line Processing Study Results

Humans and computers are similar in that source code is read line by line, top to bottom. Perhaps an intelligent pseudocode generator can apply the same strategy. In this section, we adopt a new strategy called single line processing. This technique disassembles input, multi-line programs into single lines. Then the generated single line pseudocode for each line of source code are reassembled into multi-line pseudocode. The same model from the previous section is used in this experiment. The key difference is that the evaluation dataset, StaQC, is subjected to single line processing. The results for this experiment are presented in Table 4.9.

Table 4.9: Generated multi-line pseudocode from decomposed multi-line Python source code using the DeepPseudo algorithm trained on Python source code

| ID | Source Code | Generated Pseudocode |
|---|---|---|
| 750 | from Tkinter import *<br>from Tkinter import Tk<br>nroot = Tk()<br>usernameVal = StringVar()<br>class myclass():<br>def __init__(self):<br>self.main()<br>def submitForm(self,*args):<br>try:<br>print("submitForm pressed")<br>print("name is %s" %<br>self.usernameVal.get())<br>self.usernameVal.set("")<br>except ValueError:<br>pass<br>... | from __future__ import everything into default name space .<br>from __future__ import pickle into default name space .<br>substitute decimals for root .<br>substitute decimals for op .<br>substitute the result for value .<br>define the function blankout with an argument locale .<br>return directories .<br>define the function blankout with an argument locale .<br>return directories .<br>substitute decimals for time_str .<br>substitute settings.cache_middleware_key_prefix for time_str .<br>return directories .<br>"if exception exception is caught ,"<br>... |
| 756 | pd.options.display.float_format = '${:,.2f}'.format | substitute decimals for op . |

The results shown in Table 4.9 are significantly better than the ones presented in Table 4.8. It seems that no lines of source code are skipped. However, Python syntax is still lacking. ID 750 (Table 4.9) shows that the model correctly identifies the "from Tkinter import *" as an import statement but does not realize that "Tkinter" is the name of the module to import.

The next two chapters are structured as follows. Chapter 5 will discuss the meaning and implications of our results. Chapter 6 will identify any additional work that pertains to this study, as well as future improvements that could be made to the DeepPseudo model.

# Chapter 5

# Discussion

In this work, I conducted a set of experiments to investigate the semantic quality of the pseudocode generated by the DeepPseudo algorithm. I also explored the algorithm's generalizability using diverse programming language source codes. In the first study, I replicated the original experiment conducted by Yang et al. [17]. In the second study, I explored multi-line pseudocode generation using DeepPseudo. In the third study, I examined the generalizability of DeepPseudo by using different programming languages in the training and test sets. In the fourth study, I examined the generalizability of DeepPsuedo by using training and test sets that originate from different datasets. Finally, in the fifth study, I explore improved multi-line pseudocode generation through sentence decomposition.

The results indicate that generated pseudocode quality is dependent upon the size, diversity, and the source code language of the training dataset. Furthermore, we observe significantly better results when the training and test sets originate from the same dataset. In the worst case, we observe that sentence level pseudocode quality is much better than document or semantic level quality in all studies. These studies also demonstrate that DeepPseudo performs best when the training source code language matches that of the test source code language. However, since no generalized training dataset exists, our results cannot determine if the DeepPseudo algorithm can be generalized to other programming languages.

## 5.1 DeepPseudo Replication Study

The results from this study indicate strong pseudocode quality on a sentence level, medium quality on a comparative semantic level, and low quality on a programming language semantic level. We cannot consider the quality of pseudocode on a document level here because we are only working with singular lines of source code. The results show that the DeepPseudo model often has unique but accurate interpretations of the source code. However, some semantics like operator order, function parameters, and naming are incomprehensible to the model. This suggests that the model cannot understand C++ semantics from its training set.

These results should be taken into account when considering how to train the DeepPseudo model. Transforming the

input sequence of source code into a simple embedding is not enough information to produce robust results. Instead, input sequences of source code should be converted into an embedding which includes the interdependence between multiple words in the sequence. This way, the model should be able to understand operator order, functions, and naming better. Future research into such embeddings should consider abstract syntax trees (AST) as input.

## 5.2 Multi-Line Pseudocode Generation Study

The results from this study indicate low quality on a sentence and document level, but medium quality on a comparative semantic level and low quality on a programming language semantic level. The low quality of the pseudocode on a document and sentence level show that there is an issue with the input documents. The semantic quality of the pseudocode remains unchanged from the previous study.

This study shows the importance of input document preprocessing in AI. It is possible that the inclusion of newlines and tabs to the input embeddings negatively impacted our results. Despite the ignorance of some sentences in the input document, the model still manages to generate pseudocode for multi-line programs fairly well.

## 5.3 Generalized Multi-Line Pseudocode Generation Study

The results from this study indicate low pseudocode quality on a sentence, document, and semantic level. These results show that the training source code language has a significant impact on the resulting pseudocode if the test input is from another language. In other words, the DeepPseudo model trained on one programming language does not generalize well.

These results emphasize the significance of training set and test set similarity in AI. Therefore, in order for the DeepPseudo model to be generalized to other programming languages, the training set needs to be larger and more general, or the input document embeddings should be as language independent as possible. Unfortunately, no such generalized pseudocode to source code dataset exists and the creation of one is reserved for future work.

## 5.4 Language Specific Multi-Line Pseudocode Generation Study

The results from this study indicate low pseudocode quality across the board even when the same programming language is used in the training and test sets. However, these results show that input sequence consistency is a necessary attribute between training and test sets. A limitation of this study is that the Django dataset is approximately 87% smaller than the SPoC dataset. Therefore, such a limitation negatively impacts the results of this and the following study.

These results indicate that both the training and test sets need to either be composed of single line or multi-line source code, not both. Since an AI model is usually constrained by the attributes of the training set, it is important

that the test set be processsed in a way that mirrors the training set. Alternatively, a larger Python dataset consisting of multi-line documents could be considered or constructed in future work.

## 5.5 Language Specific Multi-Line Pseudocode Generation via Single Line Processing Study

The results from this study indicate medium pseudocode quality on a document level, but low pseudocode quality on a sentence and semantic level. The acknowledgement of all source code lines by the DeepPseudo model yields a higher pseudocode quality on a document level compared to the other studies. Our results also show the hindrance known vocabulary has on pseudocode quality.

These results show the need for a diverse and sizable training dataset. Furthermore, these results emphasize the significance of programming language semantics on pseudocode quality. In other words, the DeepPseudo model needs to build a large enough list of vocabulary while understanding Python semantics. Two adjustments will need to be made in future work. Firstly, the Django dataset should be substituted with a Python dataset that is much larger and consists of code from multiple sources. Secondly, the input documents will need to be converted into an embedding that also represents Python semantics.

# Chapter 6

# Future Work

Two main improvements have been identified in this work. The first addresses the limitations imposed by few source code to pseudocode datasets available. The second addresses the accuracy of the DeepPseudo model due to the input source code embeddings.

## 6.1 Improved Datasets

This work references two datasets, SPoC and Django, that are used by Yang et al. [17, 3, 22]. From the results of this work, it is observed that accurate pseudocode generation is dependent upon the programming language of the training dataset. However, since different programming languages have different semantics, we cannot simply combine the SPoC and Django datasets together to form a larger and more generalized dataset. A simple solution to this problem is to build more datasets, yet, this comes with a huge development cost. Alternatively, the SPoC and Django datasets can be combined, assuming some preprocessing is done to the source code before it is inputted into the DeepPseudo model. This solution could benefit from an alternative input embedding that is discussed in the next section.

## 6.2 Using an Abstract Syntax Tree as the Input Embedding

Yang et al. convert input source code for their DeepPseudo model into a numerical embedding that preserves sequential information [17]. Although such an embedding can be fine for simple sequences with standard semantics, our results show that this strategy does not work well for Python source code. To combat this, a generalized embedding that preserves a programming language's semantics could be useful.

Abstract Syntax Trees (AST) are IRs traditionally used by compilers to represent source code independently from the language's syntax [38]. Furthermore, ASTs preserve semantics. Therefore, input source code sequences can be preprocessed into ASTs to generalize them. He et al. show that Transformers can use trees as input embeddings, while potentially enhancing the training process [39]. Although using ASTs as an input embedding is purely theoretical, such a solution could address the generalizability and accuracy problem found in this work.

# Chapter 7

# Conclusion

This work aimed to evaluate the semantic quality of machine generated pseudocode by studying DeepPseudo, an intelligent source code to pseudocode generator that uses AI [17]. Based on the experiments and results, it can be concluded that the semantic quality of DeepPseudo generated pseudocode is dependent upon certain shared attributes between the training and test datasets. The two most important attributes are the programming language and type of source codes, whether it is single line or multi-line source codes. The results indicate that a sizable, diverse training set and same language test set yield the best quality pseudocode.

The results from this work do show that AI generated pseudocode has potential. While the results are not as accurate as those from SMT-based algorithms [21], the quality of the results is impressive despite the small amount of training data available. Future researchers are encouraged to investigate language independent source code representations, such as ASTs, to improve dataset diversity and generated pseudocode quality while preserving programming language semantics. Ultimately, as more source code to pseudocode datasets become available, the quality of AI generated pseudocode should improve.

This work shows that the DeepPseudo algorithm is a step in the right direction for AI generated pseudocode. Compared to manual translation, an AI translation algorithm offers performance and generalization benefits. By driving down the cost to generate standardized pseudocode, it will be interesting to see whether or not computer science students learn better using it.

# Bibliography


[1] Z. Yao, D. S. Weld, W.-P. Chen, and H. Sun, "Staqc: A systematically mined question-code dataset from stack overflow," in *Proceedings of the 2018 World Wide Web Conference*, WWW '18, (Republic and Canton of Geneva, CHE), p. 1693–1703, International World Wide Web Conferences Steering Committee, 2018.

[2] F. Chow, "Intermediate representation," *Commun. ACM*, vol. 56, p. 57–62, dec 2013.

[3] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang, "Spoc: Search-based pseudocode to code," 2019.

[4] S. I. Malik, M. Shakir, A. Eldow, and M. W. Ashfaque, "Promoting algorithmic thinking in an introductory programming course," *International Journal of Emerging Technologies in Learning (iJET)*, vol. 14, p. pp. 84–94, Jan. 2019.

[5] J. B. Schafer and J. P. East, "Creating a high quality, high impact cs teacher prep program," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, (New York, NY, USA), p. 599–605, Association for Computing Machinery, 2022.

[6] G. G. Roy, "Designing and explaining programs with a literate pseudocode," *J. Educ. Resour. Comput.*, vol. 6, p. 1–es, mar 2006.

[7] Y. Kao and D. Weintrop, "Multilingual cs education pathways: Implications for vertically-scaled assessment," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, (New York, NY, USA), p. 64–70, Association for Computing Machinery, 2022.

[8] L. Silva, A. J. Mendes, A. Gomes, and G. F. Cavalcanti de Macêdo, "Regulation of learning interventions in programming education: A systematic literature review and guideline proposition," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, (New York, NY, USA), p. 647–653, Association for Computing Machinery, 2021.

[9] Q. Cutts, R. Connor, G. Michaelson, and P. Donaldson, "Code or (not code): Separating formal and natural language in cs education," in *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, WiPSCE '14, (New York, NY, USA), p. 20–28, Association for Computing Machinery, 2014.

[10] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick, "Scratch: a sneak preview [education]," in *Proceedings. Second International Conference on Creating, Connecting and Collaborating through Computing, 2004.*, pp. 104–109, 2004.

[11] R. Pausch, T. Burnette, A. Capeheart, M. Conway, D. Cosgrove, R. DeLIne, J. Durbin, R. Gossweiler, S. Koga, and J. White, "Alice: Rapid prototyping system for virtual reality," *IEEE Computer Graphics and Applications*, vol. 15, pp. 8–11, May 1995.

[12] S. Cooper, W. Dann, and R. Pausch, "Alice: A 3-d tool for introductory programming concepts," *J. Comput. Sci. Coll.*, vol. 15, p. 107–116, apr 2000.

[13] D. Weintrop, H. Killen, T. Munzar, and B. Franke, "Block-based comprehension: Exploring and explaining student outcomes from a read-only block-based exam," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, (New York, NY, USA), p. 1218–1224, Association for Computing Machinery, 2019.

[14] W. Dann, S. Cooper, and R. Pausch, "Using visualization to teach novices recursion," *SIGCSE Bull.*, vol. 33, p. 109–112, jun 2001.

[15] R. Garlick and E. C. Cankaya, "Using alice in cs1: A quantitative experiment," in *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, (New York, NY, USA), p. 165–168, Association for Computing Machinery, 2010.

[16] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. R. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. S. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *ArXiv*, vol. abs/1609.08144, 2016.

[17] G. Yang, Y. Zhou, X. Chen, and C. Yu, "Fine-grained pseudo-code generation method via code feature extraction and transformer," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, (Los Alamitos, CA, USA), pp. 213–222, IEEE Computer Society, dec 2021.

[18] B. Copus and W. P. Copus, "Pseudocode quality correlations with ultimate answer quality in cs1," *J. Comput. Sci. Coll.*, vol. 33, p. 145–150, may 2018.

[19] M. B. Wells and B. L. Kurtz, "Teaching multiple programming paradigms: A proposal for a paradigm general pseudocode," in *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '89, (New York, NY, USA), p. 246–251, Association for Computing Machinery, 1989.

[20] A. Alhefdhi, H. K. Dam, H. Hata, and A. Ghose, "Generating pseudo-code from source code using deep learning," in *2018 25th Australasian Software Engineering Conference (ASWEC)*, pp. 21–25, 2018.

[21] H. Fudaba, Y. Oda, K. Akabe, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Pseudogen: A tool to automatically generate pseudo-code from source code," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, p. 824–829, IEEE Press, 2015.

[22] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 574–584, 2015.

[23] S. Mandal and S. K. Naskar, "Natural language programing with automatic code generation towards solving addition-subtraction word problems," in *Proceedings of the 14th International Conference on Natural Language Processing (ICON-2017)*, (Kolkata, India), pp. 146–154, NLP Association of India, Dec. 2017.

[24] J. Yen, T. Lévai, Q. Ye, X. Ren, R. Govindan, and B. Raghavan, "Semi-automated protocol disambiguation and code generation," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, (New York, NY, USA), p. 272–286, Association for Computing Machinery, 2021.

[25] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, "T2api: Synthesizing api code usage templates from english texts with statistical translation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), p. 1013–1017, Association for Computing Machinery, 2016.

[26] A. L. Olsen, "Using pseudocode to teach problem solving," *J. Comput. Sci. Coll.*, vol. 21, p. 231–236, dec 2005.

[27] Y. Bosse, D. Redmiles, and M. A. Gerosa, "Pedagogical content for professors of introductory programming courses," in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, (New York, NY, USA), p. 429–435, Association for Computing Machinery, 2019.

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 6000–6010, Curran Associates Inc., 2017.

[29] Princeton University and Stanford University, "Download ImageNet Data." ImageNet. https://image-net.org/download.php (accessed May 11, 2022).

[30] S. Sharma, L. El Asri, H. Schulz, and J. Zumer, "Relevance of unsupervised metrics in task-oriented dialogue for evaluating natural language generation," *CoRR*, vol. abs/1706.09799, 2017.

[31] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, (USA), p. 311–318, Association for Computational Linguistics, 2002.

[32] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, (Ann Arbor, Michigan), pp. 65–72, Association for Computational Linguistics, June 2005.

[33] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Proceedings of the ACL Workshop: Text Summarization Braches Out 2004*, p. 10, 01 2004.

[34] R. Vedantam, C. Zitnick, and D. Parikh, "Cider: Consensus-based image description evaluation," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4566–4575, 06 2015.

[35] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, "Skip-thought vectors," in *Advances in Neural Information Processing Systems* (C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds.), vol. 28, Curran Associates, Inc., 2015.

[36] G. Forgues, J. Pineau, J.-M. Larchevêque, and R. Tremblay, "Bootstrapping dialog systems with word embeddings," in *Nips, modern machine learning and natural language processing workshop*, vol. 2, 2014.

[37] Santa Clara University, "WAVE High Performance Computing (HPC) Center." WAVE HPC. https://www.scu.edu/wave/wave-hpc/ (accessed May 11, 2022).

[38] G. Fischer, J. Lusiardi, and J. Wolff von Gudenberg, "Abstract syntax trees - and their role in model driven software development," in *International Conference on Software Engineering Advances (ICSEA 2007)*, pp. 38–38, Aug 2007.

[39] Q. He, J. Sedoc, and J. Rodu, "Trees in transformers: a theoretical analysis of the transformer's ability to represent trees," 2021.

# ColinRioux_MasterThesis_Final_unsigned

Final Audit Report 2022-06-07

| | |
|---|---|
| Created: | 2022-06-07 |
| By: | Pam Lin (plin@scu.edu) |
| Status: | Signed |
| Transaction ID: | CBJCHBCAABAAFqeXinvcX146eIT95Yt21P4FGXZlUBBX |

## "ColinRioux_MasterThesis_Final_unsigned" History

Document created by Pam Lin (plin@scu.edu)
2022-06-07 - 9:19:32 PM GMT- IP address: 24.6.72.151

Document emailed to Ihan Hsiao (ihsiao@scu.edu) for signature
2022-06-07 - 9:20:25 PM GMT

Email viewed by Ihan Hsiao (ihsiao@scu.edu)
2022-06-07 - 9:20:43 PM GMT- IP address: 66.249.84.77

Document e-signed by Ihan Hsiao (ihsiao@scu.edu)
Signature Date: 2022-06-07 - 9:21:45 PM GMT - Time Source: server- IP address: 98.47.107.87

Document emailed to David C. Anastasiu (danastasiu@scu.edu) for signature
2022-06-07 - 9:21:46 PM GMT

Email viewed by David C. Anastasiu (danastasiu@scu.edu)
2022-06-07 - 9:21:53 PM GMT- IP address: 66.249.84.67

Document e-signed by David C. Anastasiu (danastasiu@scu.edu)
Signature Date: 2022-06-07 - 9:24:00 PM GMT - Time Source: server- IP address: 129.210.115.4

Document emailed to N. Ling (nling@scu.edu) for signature
2022-06-07 - 9:24:02 PM GMT

Email viewed by N. Ling (nling@scu.edu)
2022-06-07 - 9:41:24 PM GMT- IP address: 66.249.84.71

Document e-signed by N. Ling (nling@scu.edu)
Signature Date: 2022-06-07 - 11:11:26 PM GMT - Time Source: server- IP address: 75.4.202.62

Agreement completed.
2022-06-07 - 11:11:26 PM GMT

Adobe Acrobat Sign