Department of Electrical, Computer, and Software Engineering

Part IV Research Project

Final Report

Project Number: 77

Are Machines Intelligent Enough to Create Programs Themselves?

Buster Major

Nathan Cairns

Jing Sun

04/09/2019

Declaration of Originality

This report is my own unaided work and was not copied from nor written in collaboration with any other person.

Name: Buster Major

ABSTRACT: Despite improvements in the domain of automation and artificial intelligence, most complex programs are still written by humans. A process which could produce code to an equal or greater quality than that of a human would promise to revolutionize the software discipline by removing the element of human error and freeing up developer time for more complex tasks. In this paper, we propose a novel approach for generating code to complete a partially implemented program using a type of neural architecture called Long Sort-Term Memory (LSTM). LSTMs are frequently used for music, poetry and lyrics generation but, to our knowledge, never significantly towards code generation. We have trained LSTMs on 100,000 examples of C programs and 10,000 examples of Python programs all originating from a code competition website. The models resulting from this training show potential in writing code that is syntactically correct, with up to 30% of C programs being deemed compliable, as well as signs of understanding program semantics.

LIST OF FIGURES

LIST OF TABLES

1. Introduction

1.1. Problem Description

Automatic program generation is a field of study aiming to remove the burden of writing formal, computer readable programs from the programmer, and giving this task to an automated process. As input, a programmer might give some form of high-level specification which describes the program, a set of input/output (I/O) examples or a partially implemented program to the generator. Given these input types, an automatic program generator would output a syntactically and semantically correct program which satisfies all the user's requirements, without them having to physically write the program themselves.

1.2. Motivation

Perfect automatic code generation has the potential to revolutionize the computer science community and tech industry, as it would promise to; (1) reduce development times with programs generated at computer speed, (2) reduce faults in code due to the removal of human error, and (3) streamline developer workflows, as their focus would be shifted towards more cognitively challenging, higher level tasks.

1.3. Existing Research

Current literature has broadly broken the problem space into a machine learning problem and a searching problem [1]. Search approaches taken by Sutskever *et al.*, Minton *et al.*, and Alur *et al.* involve searching through program spaces while evaluating each partial program by some heuristic, a process which can be both computationally intensive and difficult to implement [2, 3, 4]. Learning approaches involve specialized machine learning processes to identify patterns in high-level specifications or I/O examples when mapped to resulting complete programs. Typically they involve some form of machine translation from one specification to another, such as what Delvin *et al.*, Sing *et al.*, Balog *et al.* and Ling *et al.* demonstrate [5, 6, 7, 8].

1.4. Research Intent

In this project we intend to expand the existing research of partial program completion. This is where a partially completed program is fed into a model, and the model works to complete the program to semantic and syntactic

correctness. We intend to explore the possibility of using a specialized type of neural network for this; a long short-term memory architecture (LSTM). LSTMs are proven to be good at memorizing long-term dependencies across sequences of tokens [2], and hence theoretically lend themselves to generating correct complex output patterns, such as high ceremony languages like computer code. This is a machine learning project, where program examples must be pre-processed before feeding into the model for training and generation. Furthermore, we aim to evaluate the performance of our model in generating syntactically and semantically correct code. These results will be compared with literature from the automatic program generation domain, as well as research from the LSTM sequence generation domain, in order to evaluate the LSTMs utility in generating useful code and promise for future research.

Table 1: Division of work between Buster Major and Nathan Cairns

Task	Nathan Contributed	Buster Contributed
Research	X	X
Design	X	X
Tokenization		X
Training		X
Generation	X	X
Evaluation	X	
Comparative Analysis	X	X

1.5. Report Structure

Section 2 discusses existing literature surrounding automatic code generation and the current use of LSTMs as well as a discussion of gaps in the research and how our solution targets aspects of this. Section 3 discusses our proposed solution at a high design level, as well as how it will go about answering our proposed research questions. Section 4 describes our implementation including what technologies, datasets and data preprocessing techniques we have used. Section 5 details our methods of evaluation, our results and their implications of the project. Finally, section 6 summarizes the project and highlights research contributions.

2. Literature Review

2.1. Program generation by learning

The learning approach to automatic program generation typically harnesses the power of recurrent neural networks (RNNs) to translate a set of input / output (I/O) examples to a functional program [5, 9]. The choice

of using I/O examples as a program specification is popular because it alleviates the need for a programmer to write a formal program description to use as input to the program generation model, which is often more difficult than writing a program alone [5, 10, 11]. Delvin *et al.*, Singh *et al.*, Balog *et al.*, Sutskever *et al.* and Ling *et al.* all employ similar two stage neural architectures for training; one network for transforming a variable length specification into a constant length latent vector (encoder) and a further network to map this specification vector to a program (decoder) [5, 6, 7, 2, 8]. Encoders and decoders are typically built from LSTMs [2].

2.2. Program generation by search

Generating a program through searching requires iterating through a (possibly infinite) program search space and checking candidate programs through some validation mechanism [9, 12, 4, 6]. Because it is desired to output programs which are dynamic and expressive in language, program search spaces can become very large very quickly, so much research in this space focuses on reducing this [6]. Alur *et al.* use version space algebra to drastically reduce their search space by defining the program specification as a partially implemented program, the writer of which can assert *holes* in the functionality left up to the synthesizer to search for and implement [4]. Perhaps the most significant existing search based technology is the Microsoft Excel 2013 feature *FlashFill* which allows users to define examples of string manipulations/transformations by defining I/O, the program then generates a regex like program which satisfies the users specifications [5, 9]. The result of Singh *et al.*'s research is a tool called *SKETCH* which is able to fill in holes of a partially implemented program. A programmer is able to write code declaratively and let the synthesizer find a suitable implementation by searching, the process of which also takes advantage of SMT solver technologies [12].

2.3. Long Short-Term Memory Machine Learning Architectures

Long Sort-Term Memory (LSTM) architectures are a form of neural network which are good at remembering long term dependencies. If a network generates a sequence of output tokens, it can remember the context of information it received as input and arbitrary number of timesteps ago. This makes it a desirable technology for use in automatic language generation, as it is able to keep track of 'where it is in a sentence' and what characters/words would be appropriate to follow [13]. Potash *et al.* showed that LSTMs can generate rap lyrics

with coherent rhyme schemes, showing their ability to remember rhyming context [14]. Oliviera *et al.* similarly showed the LSTM's ability to learn a typical poem structure and generate lines belonging to either haikus or sonnets which would fit the flow of the required format [13]. Choi *et al.* illustrate performance in the music writing domain, where LSTMs are able to remember the key, style and pitch of previous notes generated to write a relatively cohesive musical piece [15].

2.4. Summary of Findings

Program generation can be typically seen as a learning exercise where some model is induced which can translate some program specification to a program [7, 5, 2]. Additionally, LSTMs are frequently used in both model generation and search approaches as a normalization tool [2, 7, 5, 9], mainly because of their impressive ability to retain long-term dependencies, which is important for formal and highly structured languages such as programming languages. A gap we have identified in this research, is using LSTMs to generate code itself, rather than an intermediary normalization function where they are usually used [2]. The LSTM's ability to generate structured, cohesive output is applicable to the typically formal and sensitive structure of programming code. This approach is novel as it would investigate the feasibility of an LSTM learning the syntax of a programming language as well as its ability to learn anything about the 'meaning' of a line of code by syntax alone. The ubiquity of LSTMs across automatic program and natural language generation research illustrates the recent success in the technology, which makes it a promising choice for further research. Research in program generation with an LSTM would be analogous to the natural language generation of Potash *et al.* and Oliviera *et al.*'s research [14, 13] except in a language domain with greater restrictions on syntax and semantics. Additionally, like Singh *et al.* and Alur *et al.*, this approach would investigate partial program completion [6, 4], but with a different underlying synthesis technology, expanding this research space also.

3. Proposed Research Solution

We propose a tool which takes as input a program with *n* last lines missing, and is able to generate *n* lines to append to the end of the program. An LSTM will generate these lines, and ideally, it will have understood the syntactic context of the program it was given, where it will then be able to generate a syntactically valid

addition to the program. Additionally, it will hopefully have recognized some programmatic patterns so as add to the program semantically as well. These patterns could be recognized in the form of keywords, where (as an example in the python language), if the model has an incomplete *try* statement fed in, then it would know to eventually complete the statement with an *except* or *finally* statement.

Our research investigates the extent to which LSTMs have the potential to understand the syntax and semantics of a given program, and to what level of code quality it can generate from this. Our approach could be considered a streamlined approach of Sutskever *et al.*'s research [2], where the transformational aspect is removed and the LSTM's function is distilled. Our research also extends the LSTM text generation research domain, as research already exists in natural language [13, 14] and music generation [15], but to our knowledge none which seriously investigate code generation. With our research aiming to investigate this combination of research gaps, we feel this area is rather novel. Because of this novelty, we aim to investigate the feasibility of this approach by highlighting strengths and weaknesses. An aspect of this evaluation will be an investigation into the change in quality of code produced across different language paradigms, specifically, a dynamically typed loose syntax language and a strongly typed strict syntax language. Furthermore, forms of text pre-processing distinct from what is investigated in [13, 14, 15] must be pioneered, due to the differences between code and natural language/music, such as formality, strictness and scope.

The LSTM's strengths of strong long-term memory retention will ideally be harnessed when used for code generation. As programming languages are typically highly structured in the sense that certain language constructs are only valid to follow or precede other language constructs, the LSTM's ability to apply knowledge of it's given context to generating characters should aid it in outputting synthesized code which adheres to syntax requirements. For example, in the language C, an LSTM would ideally learn that a closing '}' is not valid syntax unless the context is such that an unclosed opening '{' exists somewhere previously. The LSTM's dependency retention may also carry over to program variables, where it may learn that the declaration of an indexing variable when enumerating a collection in some language is typically used to index the collection itself, specifically '*myCollection[i]'*.

3.1. Research questions to be answered

We expect our research to indicate whether the LSTM's success in generating speech or music can translate to a far stricter and more structured domain like computer code. We also look to investigate the LSTM's viability as a program synthesis technology with respect to current program synthesis techniques.

4. Implementation

4.1. Choice of technologies for implementation

We have chosen to implement the project in Python due to its popularity in machine learning projects. Additionally, it is a good language for scripting and programming I/O functions with little overhead. We have chosen the Tensorflow machine learning library for harnessing an LSTM, an open source library for large-scale machine learning developed by Google [16]. We selected this technology because of the reputability of the creators and the ease at which the tools can be harnessed with very little lines of code. Additionally, Tensorflow has a wealth of documentation and tutorials on implementing LSTMs which reduces the overhead of us understanding the technology. We would set up LTSMs with default hyperparameters defined in an LSTM tutorial [17] found on the Tensorflow website. These hyperparameters would be acceptable given the investigative nature of the project.

4.1.1. Training Environment

We had access to a Microsoft Azure Cloud Services VM for development, training and evaluation. The VM ran Linux and had to be accessed through SSH. Using the VM for the heavy computation provided the benefits of stability and reliability. Furthermore, the VM had access to a plentiful amount of resources, meaning we would be able to harness more computational power to train and evaluate. Unfortunately, VMs typically do not have access to GPUs, which are useful for speed in machine learning with Tensorflow. Non-GPU training can be significantly slower than GPU training, meaning we had to limit the number of training cycles we executed to fit within a training timeframe of around 3-4 days.

4.1.2. Python development tools and libraries

We would use packages *ast* and *Clang* for Python and C tokenization respectively, these will be further

discussed in 4.4.2 and 4.4.3. Other notable libraries include library *pylint* for evaluating generated Python code executability and *Python-Levenshtein* for evaluating generated code similarity. These packages can be found by name on the Pip Python package repository. In addition to these, the standard GCC compiler was used for evaluating generated C code executability [18].

4.2. Implementation design

The main functionality of training, generating and mass evaluating our project is found in 3 main Python driver scripts (illustrated respectively as *train.py*, *generator.py* and *evaluate.py* in Fig. *1*.) Notably, *generator.py* is independent from data and it therefore easily portable as an independent tool.

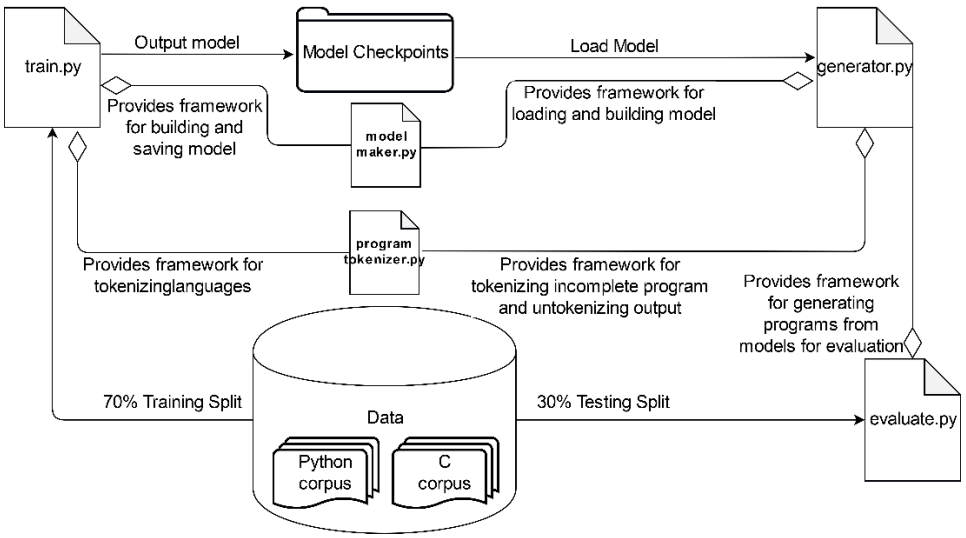

Fig. 1. Design view of implementation

4.3. Choice of programming languages to model

It is important for us to be able to evaluate the proposed program synthesis technique across more than one language to assess if there is a language paradigm better suited to the technique. In this project we were able to integrate and fully evaluate 2 different programming languages, a dynamically typed, interpreted language and a strongly typed, compiled language. The dynamic language we chose was Python. A language relaxed enough to forgive errors relating to variable definition or incorrect usage of types [19] would mean that the code our model would produce would be under less scrutiny by the Python interpreter. We chose C as a strongly typed language because, like Python, it is low ceremony and its syntax is basic. Although C is strongly typed and compiled, its simplicity makes it a good choice for piloting the LSTM technique, because a model has less

syntactical overhead to generate.

4.4. Tokenizing the languages

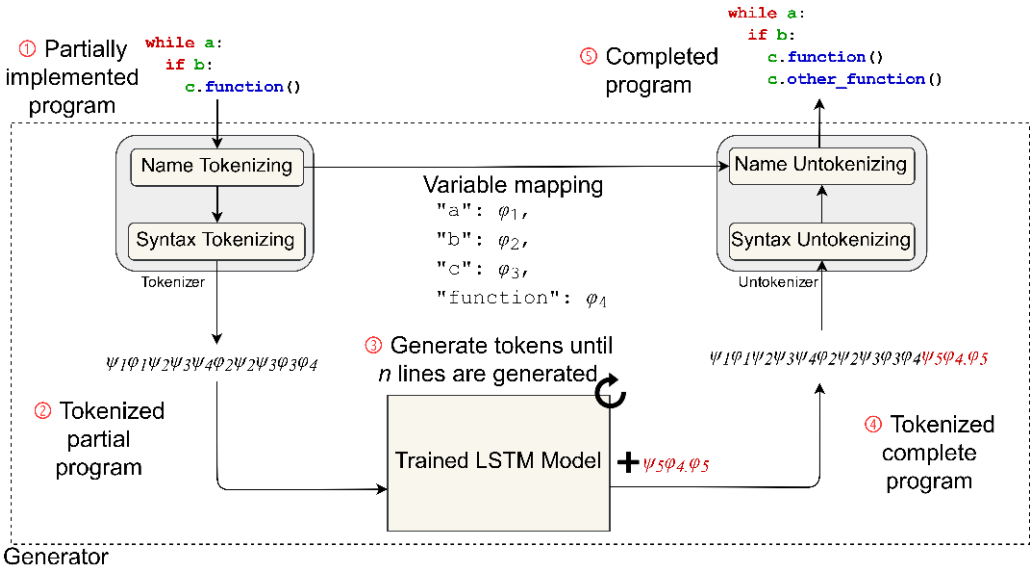

Fig. 2. Generalized view of code generation workflow

LSTMs take as input a sequence of discrete tokens, and output further tokens given the model's training history. The minimization of information that is fed into an LSTM can dramatically improve its performance [20, 21]. For example, an LSTM, when needing to generate a *for* keyword will not need to generate the *f-o-r* characters sequentially but only concern itself with a single arbitrary *for* token. A high-level workflow of the proposed tool is presented in Fig. 2, which outlines the modules of the design and shows the flow of the data. Step 1 of Fig. 2 illustrates an input program string being converted into a series of tokens. Step 2 shows tokenization, which is split into a (1) naming tokenization stage, and (2) a syntax tokenizing stage (discussed in 4.4.1). This mapping is stored in memory while the tokenized program is fed into the LSTM as input and is used to generate the next *n* lines in step 3. Generated content from the LSTM is joined to the original tokenized program in step 4, the result of which is detokenized through a simple replacement process in step 5.

4.4.1. Syntax vs named structure tokenization

Syntax tokenization effectively maps keywords to tokens statically for all program instances. An LSTM will therefore be able to form a strong understanding of the context in which these tokens are typically used. However, the same cannot be said for other program structures like programmer named variables, types and

functions (which will be referred to as named structures). Each program instance will likely have differing name and structural choices which drastically change the meaning of a variable, function or type's semantics. This has two major implications for named program structures; (1) variables which are semantically similar may have completely differing names across programs, and (2), variables with the same name across multiple programs may have completely differing semantic meanings. A non-name-based mapping scheme would remove these problems and make the training vocabulary smaller [22].

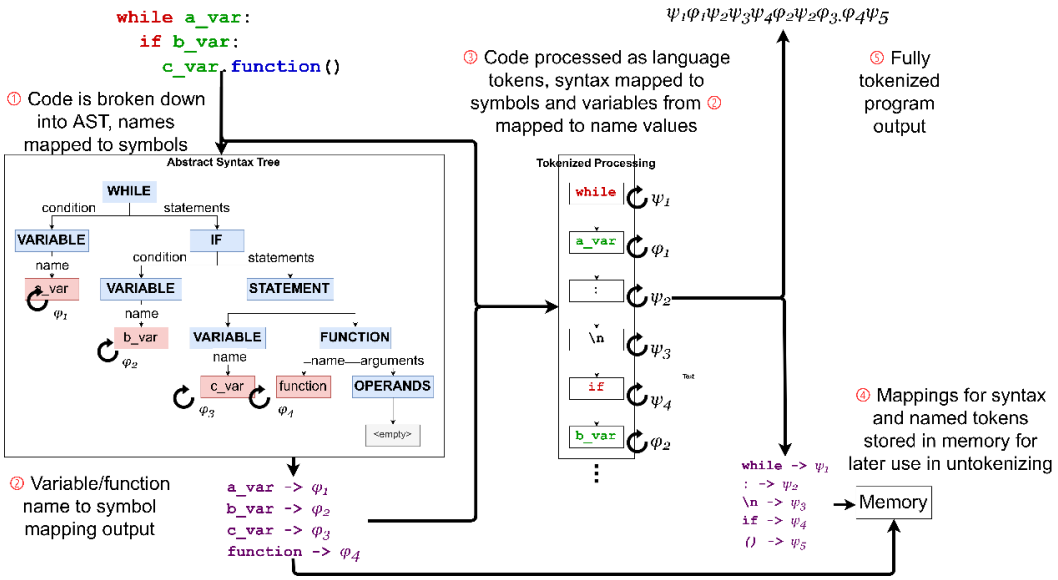

Fig. 3. Detailed workflow of program tokenization

An example of a non-name based mapping scheme is to assign tokens to the named structures of a program in an ordered fashion, where the first occurrence of a named structure is assigned symbol φ_1, the second is assigned φ_2, etc. While this approach is simple in its generalization of named structure declaration order determining token instance, it is adequate due to the project's experimental nature. Issues resulting from this assumption will likely include models struggling to use variables in correct contexts due to token order being the only constant across programs. Fig. 3 details the named structure and syntax construct tokenizing phases. Step 1 shows a program string being broken down into an Abstract Syntax Tree (AST) format unique to its language, where named structures can be distinguished from other program tokens. An AST is useful because it can identify types of language tokens at a granular level. Step 2 shows a mapping being produced from these named structures to a collection of ordered tokens which is kept in memory. Step 3 shows the original program

string being processed token by token and swapped with symbols from the syntax token set or the named structure token set where applicable. In step 4 these token sets are further retained in memory for de-tokenizing purposes, and step 5 shows the completed tokenized program returned as output.

4.4.2. Implementation of Python tokenization

Python AST analysis is done using the standard Python *ast* library [23], a default package that comes with the standard Python distribution. This package was chosen because we could be confident of its correctness and robustness due to its origins in the Python Software foundation. Syntax tokenization of Python programs would be done by breaking program strings into a series of tokens using the *tokenize* Python module, which acts as a lexical scanner for the language. Both *tokenize* and *ast* libraries were required because *ast* couldn't provide functionality for manipulating program tokens robustly and *tokenize* couldn't identify named constructs at a fine-grained enough level. Python tokenization also includes the transformation of function definitions where unnecessary syntax is removed (illustrated in Fig. 4), simplifying LSTM input. Additionally, Python's functional use of indents is simplified in Fig. 5, where *newline* and *dedent* (opposite of *indent*) tokens are added to preserve scope and statement discretion. This allowed to indents to be removed entirely.

```
def my_function(a, b, c):     def my_function a b c:
```

Fig. 4. (Left) Un-simplified function definition. (Right) Simplified function definition

```
for a in my_list:          for a in my_list: φnewline a = '12345'
    a = '12345'            φnewline φdedent b = a
b = a
```

Fig. 5. (Left) Un-simplified indenting scheme. (Right) Simplified indenting scheme through tokens

4.4.3. Implementation of C tokenization

Unlike Python's split of tools to traverse the AST and transform the code, we were able to use a single tool *Clang* for both traversal and transformation of C. *Clang* is used as a tooling software for building compilers for the C language family [24], hence it can provide fine grained analysis of program tokens. *Clang* was chosen because of its reputability as an established piece of software. Due to C's usage of curly braces denoting scope and semi-colons denoting statements, new lines and indents can be removed without risk of misinterpretation. Other additions to the C tokenization process from that outlined in Fig. *2* and Fig. *3* are trivial.

4.5. Choice of data sets

Because it is often useful to limit the scope of code being generated when piloting a new approach [5, 9, 7], it is important that our program datasets reflect a narrowed domain of application, which may allow an LSTM based model to find clearer programming application patterns in the data.

4.5.1. Codechef Competitive Programming dataset

We have used a multi-language dataset sourced from the Codechef competitive programming website [25]. All programs in this corpus are user submitted programs for simple programming practice questions. There are several advantages to using this dataset over other datasets; (1) both Python and C program corpuses are from the same domain, hence comparable, (2) both Python and C programs are simple, (3) the scope of programs trained and therefore generated is constrained. Notably, there are approximately 10 times the number of C program examples (~100,000) than Python program examples (~10,000). Additionally, the evaluation of the approach is tested on models trained on 10%, 50%, 75% and 100% of their respective training data. This means that the C model trained on 10% of the available data and the Python model trained on 100% of the available data are comparable.

5. Evaluation

We have compared the models trained on Python and C code across a series of metrics to establish numerical points of difference between an LSTM's ability to generate the languages. This allows us to understand what forms of code this LSTM based modelling technique may be suited for. Additionally, we have empirically evaluated a set of sample generations to further solidify the data presented in these metrics.

5.1. Results

Fig. 6 and Fig. 7 each depict a metric which samples a set of models trained on 100%, 75%, 50% and 10% of data accessible to them. The range of models trained on differing data splits work to forecast the potential for further improvement as models have access to more data. For the following sections, the term 'equivalent' models will refer to the 100% trained Python model and the 10% trained C model, each being trained on ~10,000 examples. Blank lines are not generated by the models.

5.1.1. *Syntactic program correctness*

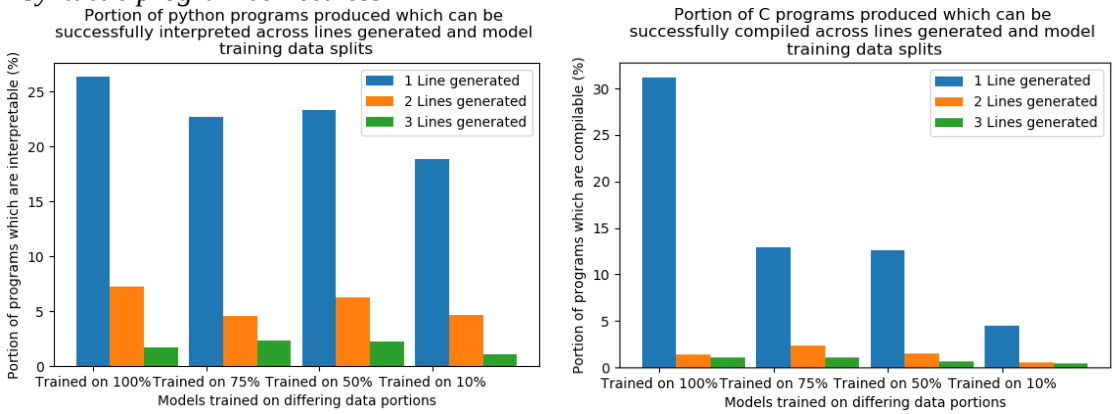

Fig. 6. Program executability percentage when asked to generate 1, 2, and 3 lines for Python (Left) and C (Right)

The syntactic correctness of a Python program is determined by running an output script through a linter and identifying critical errors received. C syntactic correctness is determined by running programs through a compiler and checking if the compilation fails. As illustrated in Fig. 6, the syntactic correctness of programs generally increases the more data the model has access to, and neither model appears to plateau. Comparing equivalent models, we can see Python reaches a higher syntactic accuracy of around 25% vs. the C model's 5%. This is logical given C's stricter compilation process. Additionally, it is promising that the C model's syntactic correctness steadily increases with more training, indicating the Python model could improve significantly from 25% if provided more training data. As expected, this syntactic correctness for both models falls sharply as the models are asked to generate more code (by generating more lines), although the C models suffer more in this respect. This trend works to reaffirm C's stricter typing / compilation process.

5.1.2. *Accuracy of model keyword/variable guessing*

The accuracy of a model correctly predicting the next language keyword is determined by checking the next language keyword in the removed line of the test program, and cross checking this with the next language keyword in the generated lines for that program. If neither original nor generated lines have keywords, then the example is not counted. Likewise is calculated for variable occurrences, with the added condition of if the original program introduces a new variable on these lines, then the equivalent for a generated line would be to introduce a new variable with an aliased name such as *temp0*. If this new variable has not been introduced into the program earlier then the variables are considered equivalent. As illustrated in Fig. 7, the C model is

approximately equal to its Python counterpart in keyword prediction accuracy, with both values around 20%. Additionally, the further trained C model reaches an accuracy of 40%.

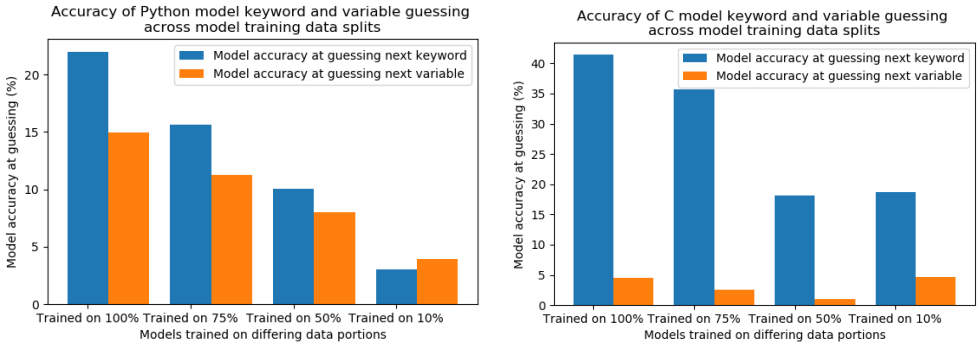

Fig. 7. Model accuracy of guessing next keywords and variables for Python models (Left) and C models (Right)

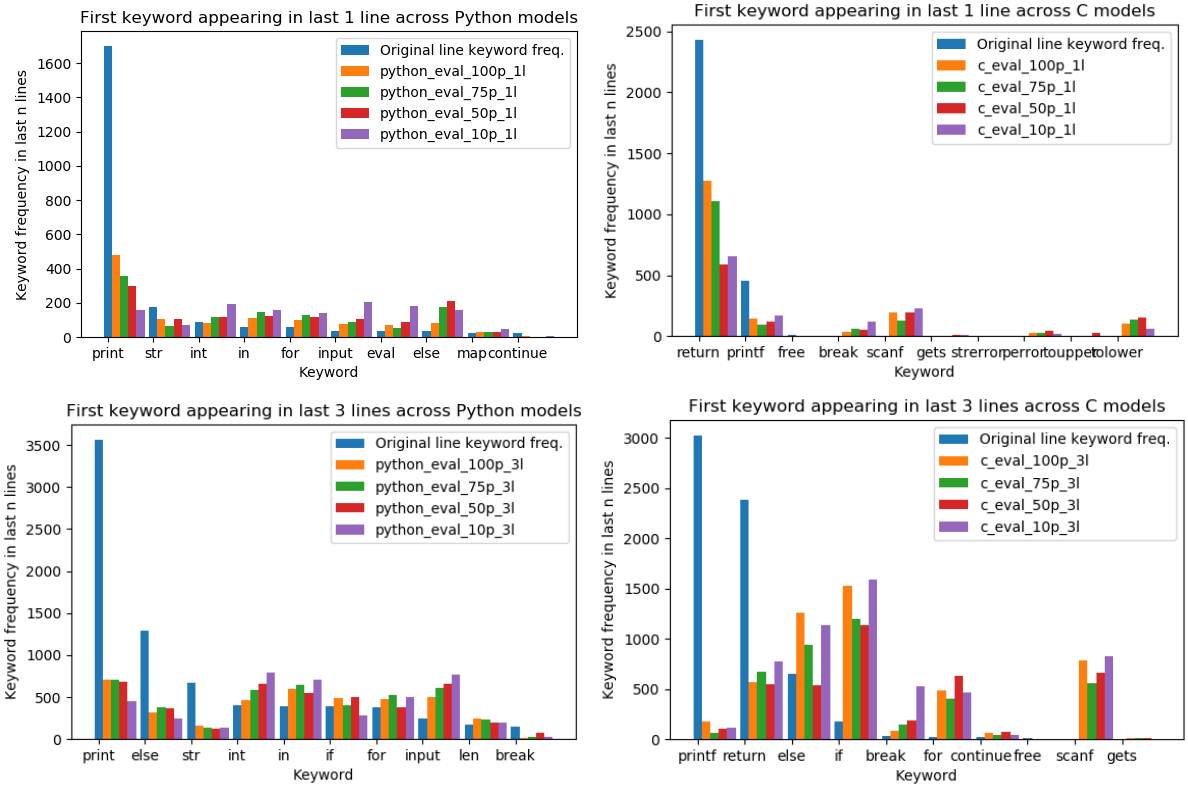

Fig. 8. First appearing keyword frequency in original files and model training splits across Python models tested on generating last 1 line (Top-Left), C models tested on generating last 1 line (Top-Right), Python models tested on generating last 3 lines(Bottom-Left) and C models tested on generating last 3 lines (Bottom-Right)

The insight provided by Fig. 8 adds further intrigue to the models' keyword guessing ability, illustrated by the difference in first keyword frequencies when asked to fill in the last 1 line versus the last 3 lines. (In the key,

<*int*>*p* refers to the training data split, e.g. *50p* is 50%). As the models are asked to generate lines *earlier* into the program (last 3 lines) their keyword prediction skews less toward *print*, and more toward *in*, *if*, *for*, etc. Shown in the C model keyword frequency splits, keywords like *if* and *else* appear far more frequently when asked to generate lines earlier in the program. The context of the code is generally different in the last line of a program versus the third to last line, and as the model reflects knowledge of this difference in context, it implies that the models have begun to understand the appropriateness of certain keywords given a certain program context. In other words, they have begun to understand program semantics, if only slightly. However, this trend is fairly weak and it is clear that the models still randomly guess fairly frequently. In the top-left Python figure in Fig. 8, for example, while the *print* keyword is the most frequent occurrence in the original lines removed at around 1700 occurrences, the trained model only guesses around 500 occurrences. Although a clear trend can be seen of each model's keyword prediction frequencies shifting closer to the true keyword frequencies over training data splits, more data would be required to reach correct keyword prediction frequencies. Additionally, the C model, although exposed to 10 times the number of training examples still severely underestimates the frequency of C's most popular ending keywords; *return* and *printf*.

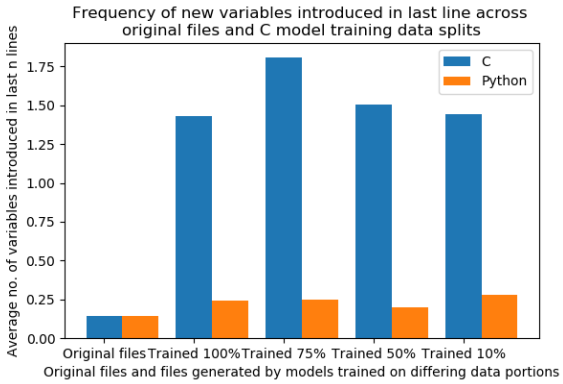

Fig. 9. Frequency of new variable introduction in last line of program across Python and C models

With respect to variable guessing accuracy displayed in Fig. *7*, the data from the Python model can be seen to exceed the accuracy of the equivalent C model, with a value of 15% accuracy vs. 5% accuracy. It is also important to note that, unlike the Python model, the C model does not appear to improve in variable prediction as the models are exposed to more training data, suggesting that it may be at a plateau with no room for

improvement. A likely cause of this is illustrated in Fig. 9, where all C models are shown declaring far higher rates of new variables in the last *n* lines than in evaluation data its Python counterpart. While we are unsure of why the C model does this, we are confident it is the reason for the C model's variable prediction being so low, as the model clearly predicts that new variables appear far more frequently than they really do.

Table 2: Keyword and variable stats compared with random guessing for C and Python models

	Python Model			C Model		
	Random guessing	Generated	Δ	Random guessing	Generated	Δ
Keywords	0.93%	22.00%	21.07%	1.69%	18.72%	17.03%
Variables	9.72%	14.93%	5.21%	7.06%	4.62%	-2.44%

Table 2 displays the differences in model keyword/variable prediction accuracy versus the guessing accuracy if the models were randomly choosing. Random keyword guessing is calculated by the chance of randomly guessing any of the reserved keywords for each language. Random variable guessing is calculated by the chance of randomly guessing from the average number of distinct variables not included in the last *n* lines, with 1 added to account for a newly declared variable. While C and Python models are better at keyword guessing than random, we believe a factor in this performance is that the distribution of most frequently used keywords in each language is skewed heavily towards a small number of frequently used ones (like *printf, return* for C and *print, str* for Python). This preference for learning the more frequent keywords would likely result in a higher frequency of correctly predicted variables across both languages. However, we do not believe the entirety of the models' strategy is to randomly guess the most frequent keywords, as we know there is an element of context it takes into consideration when making its guess, as discussed earlier.

Interestingly, the Python model's variable prediction is better than random guessing. Unlike keywords, this statistic cannot be influenced as much by skewing of variable frequency as variable-token mapping is fairly random across program examples. Therefore, the fact that the Python model is able to show an improvement here is indication that it has started to understand the patterns in which variables are declared and subsequently used. Alternatively, the C model shows a prediction rate worse than if it had randomly guessed all variables, further supporting the claim that for whatever reason the model was influenced to generated new variables far more frequently than what was represented in the data.

5.1.3. Empirical Evaluation

We have evaluated several lines of generated code across five partially implemented programs from the evaluation data set for each language model against a set of criteria. The partial programs are simple examples like calculating the Fibonacci sequence. Three non-whitespace non-bracket lines have been removed from each program, three lines have been generated and these are evaluated against a set of criteria. They are; (1) quality of usage of new variables (variables introduced by the model), (2) quality of syntactic styling, (3) use of variables of certain types in correct context, (4) overall semantic quality. Each criterion is rated one of *poor*, *average* or *good*. This evaluation was done using the C model trained on the full 100,000 examples, and the Python model trained on 10,000 examples.

Table 3: Scores for models across given criterion for performance in code generation across code examples

	Python Model					C Model				
Program	GCD	LCM	Factorial	Gnome sort	Merge sort	Is Prime	Recurse	Array max	Transpose	Average
New Variables	poor	poor	poor	good	good	poor	average	poor	poor	poor
Syntax/Styling	good	poor	average	poor	good	good	good	good	average	poor
Var. type use	average	average	average	good	average	good	average	good	poor	average
Semantics	average	average	poor	average	average	poor	average	poor	poor	poor

The empirical evaluation results displayed in Table 3 do not appear to significantly favor either language, despite the C model being exposed to more training data. A recurring theme was found where the models tended to use new variables without first declaring them. This was especially frequent when the partially implemented program passed to the model had very few variables to begin with. We believe this is because of the way we have tokenized the names structures in each language. For example, a partial program has one existing variable, but the model has been trained to be aware of several variables per program, meaning the variables it generates are less likely to be the one variable that has been declared. We would have hoped that the model would have learned to declare new variables before use and try to reuse existing variables more, but empirically we see little evidence of this. Additionally, both language models show a tendency for using variables and functions interchangeably. While this is in principle possible in a dynamically typed language like Python, models rarely cast or redefine to make this changeability appropriate. Neither model shows much evidence of an understanding of high-level program semantics, as lines generated appear to be randomly chosen language constructs which do not help the overall goal of the program. Our empirical evaluation can

be summarized by saying; the models appear to generate well-formed code but use of variables and semantic correctness are not generated to an acceptable standard.

5.2. Discussion

We believe the results discussed indicate there is potential for significant improvement in the models' ability to identify a program's next keywords, variables and overall executability. Given the improvement over model generations as they are trained on more data discussed in 5.1.1 and 5.1.2, we believe having access to a larger dataset will allow us to improve the models performance on these metrics without significant alteration to the approach. Given the results we have, we believe there is potential in LSTMs learning the syntax of a programming language well, but there appears to be limitations in the models' ability to understand semantic context such as variable scope and variable typing (despite showing slight signs of understanding program context). While the models' tendency to use variables in a more dynamically typed fashion better lends itself towards Python programming, the current approach still appears limited when it comes to using these named constructs correctly. A likely cause of this is the way named structures are tokenized; both function names and variable names are mapped to the same set of tokens, and the model clearly struggles to discern the difference between their usage. With a significantly larger data set or an alternative tokenizing method, this issue can likely be reduced. Furthermore, the models' tendencies to use variables when out-of-scope implies they have not learned this concept either. More data, training or a way of embedding the variable scope information to the variable's tokenized form may be able to address this issue.

In comparison with our model's output code quality, Ling *et al.* employed an approach of plugging missing statements into templated code to guarantee well-formedness [8], hence had a 100% syntactically correct output and almost guaranteed semantically correct output, far higher than our results. However, an approach like this for this project is likely infeasible due to their research being far tighter in scope. The performance of our approach reaching approximately 27% Python and 32% C code well-formedness compares poorly to Delvin *et al.* who were able to reach 90% syntactic and semantic accuracy in their program generation [5]. This disparity in accuracy is likely due to lack of access to data and their research being tighter in scope. Oda

et al. reported a 50% semantic acceptability in comments generated when evaluated empirically [11], and referring to Table 3, if we are generous and say that a score of average or above in all four criterion is considered semantically acceptable then our approach again falls short with a 20% semantic acceptability. Oda *et al.*'s research also focuses on machine learning interpreting the meaning of given code, however their output is far more high level and semantically relaxed than that of code, so this difference in accuracy should be expected. Our approach has results comparative to Choi *et al.*'s, where their use of pure LSTMs did learn to pick up on basic music patterns but struggled to learn long term patterns in drum passages [15]. This is analogous to our research which learned program syntax well but failed to learn a significance in declaring variables before use or variable scope. They attributed this issue to the turning of their hyperparameters, which prompts the need for our LSTM setup to be expanded also. The performance of our model has been outperformed by that of Potash *et al.*'s use of LSTM in generating rap lyrics, where rhyme density and novelty of the model's output were both analogous to their evaluation data's [14], while our approach struggles to reach the same semantic accuracy as its evaluation data.

6. Conclusion

We have evaluated a novel way of generating code for partially complete programs in Python and C using an LSTM neural architecture and tokenization system. The approach of using a distilled LSTM for means of generating code is somewhat novel in the field, as many other researchers use it as a complementary tool in their processes. We have shown that the technique shows promise for understanding language structure as 27% of Python code generated is deemed interpretable and 32% of C code is deemed possible to compile. In addition to this, model predictions for next keyword reach around 40% accuracy, while variable predictions reach as high as 15% accuracy. These results are promising given the amount of data we had access to for training. A weakness in the approach is the model's understanding of program semantics and variable use. While it performs worse in semantic quality than similar research in the program generation research space, this is to be expected given the relatively broad scope of the programs generated in this project as well as the simplicity of the LSTM setup. Additionally, our evaluation does show evidence that the models have begun to learn program semantics, if only slightly. When compared to research in the LSTM text/sequence generation

research space, results are closer to yet still underperform. We believe this is because the increased ceremony of code compared to music or natural language, as well as a lack of data and minimal LSTM hyperparameter tuning. The results outlined here serve as a benchmark for research looking to improve upon the pure LSTM-based code generation approach. Additionally, this research serves as an extension to the current LSTM language/sequence generation field by pushing the technology further in complexity, ceremony and length.

6.1. Future Work

It would be beneficial to the model's semantic and syntactic performance to encode information on variable scope and type into its tokenization. This would likely improve on its tendencies to use variables when out of scope or inappropriately given its type. Furthermore, comparing the model's performance in generating another language paradigm such as functional programming would work to drastically increase the scope of research surrounding this approach. As functional programming languages do not typically employ variables, it would not suffer from current issues relating to variable usage and would be able to be more closely comparable to Delvin *et al.* and Feser *et al.*'s research which also generate functional programs [5, 9]. Experimenting more with the tuning of LSTM hyperparameters may also lead to greater results, as shown by Choi *et al.* [15]. Finally, more data would likely improve performance as forecast in our evaluation.

Acknowledgements

We would like to thank our supervisor Jing Sun and doctoral candidate Chenghao Cai for their expert opinion and guidance throughout this project. I would also like to acknowledge partner Nathan Cairns for his diligence and efforts throughout this experience.

References

[1] R. Simmons-Elder, A. Miltner and S. H. Sebastian, "Program Synthesis Through Reinforcement Learning Guided Tree Search," *ArXiv,* vol. abs/1806.02932, 2018.

[2] I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," *CoRR,* vol. abs/1409.3215, 9 2014.

[3] S. Minton and S. R. Wolfe, "Using machine learning to synthesize search programs," 1994.

[4] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak and A. Udupa, "Syntax-guided synthesis," Oct 2013.

[5] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-R. Mohamed and P. Kohli, *RobustFill: Neural Program Learning under Noisy I/O,* 2017.

[6] R. Singh and P. Kohli, "AP: Artificial Programming," Jan 1, 2017.

[7] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin and D. Tarlow, "DeepCoder: Learning to Write Programs," *CoRR,* vol. abs/1611.01989, 11 2016.

[8] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang and P. Blunsom, *Latent Predictor Networks for Code Generation,* 2016.

[9] J. K. Feser, S. Chaudhuri and I. Dillig, "Synthesizing data structure transformations from input-output examples," *ACM SIGPLAN Notices,* vol. 50, pp. 229-239, 6 2015.

[10] S. R. Branavan, H. Chen, L. S. Zettlemoyer and R. Barzilay, *Reinforcement Learning for Mapping Instructions to Actions,* 2009.

[11] 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 (T)," Nov 2015.

[12] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia and V. Saraswat, "Combinatorial sketching for finite programs," *ACM SIGARCH Computer Architecture News,* vol. 34, p. 404, 10 2006.

[13] H. Gonçalo Oliveira, R. Hervás, A. Díaz and P. Gervás, "Adapting a Generic Platform for Poetry Generation to Produce Spanish Poems," in *Proceedings of 5th International Conference on Computational Creativity*, Ljubljana, 2014.

[14] P. Potash, A. Romanov and A. Rumshisky, "GhostWriter: Using an LSTM for Automatic Rap Lyric Generation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, Lisbon, 2015.

[15] K. Choi, G. Fazekas and M. Sandler, Text-based LSTM networks for Automatic Music Composition, London: arXiv preprint arXiv:1604.05358, 2016.

[16] Tensorflow, "Introduction to Tensorflow," [Online]. Available: https://www.tensorflow.org/learn. [Accessed 12 Septe,ber 2019].

[17] Tensorflow, "Tutorials - Recurrant Neural Networks," Google, [Online]. Available: https://www.tensorflow.org/tutorials/sequences/recurrent. [Accessed 24 09 2019].

[18] Free Software Foundation, "GCC, the GNU Compiler Collection," Free Software Foundation, 19 08 2019. [Online]. Available: https://gcc.gnu.org/. [Accessed 24 09 2019].

[19] Python Software Foundation, " Python » 3.3.7 Documentation » The Python Tutorial » 8. Errors and Exceptions," 19 September 2017. [Online]. Available: https://docs.python.org/3.3/tutorial/errors.html. [Accessed 11 September 2019].

[20] B. Heinzerling and M. Strube, "BPEmb: Tokenization-free Pre-trained Subword Embeddings in 275 Languages," *CoRR,* vol. abs/1710.02187, 2017.

[21] P. Wang, Y. Qian, F. K. Soong, L. He and H. Zhao, "A Unified Tagging Solution: Bidirectional {LSTM} Recurrent Neural," *CoRR,* vol. abs/1511.00215, 2015.

[22] H. Liao, G. Pundak, O. Siohan, M. K. Carroll, N. Coccaro, Q.-M. Jiang, T. N. Sainath, A. Senior, F. Beaufays and M. Bacchiani, "Large Vocabulary Automatic Speech Recognition for Children," in *Interspeech*, Dresden, 2015.

[23] Python Software Foundation, "ast - Abstract Syntax Trees," 11 September 2019. [Online]. Available: https://docs.python.org/3/library/ast.html#module-ast. [Accessed 11 September 2019].

[24] The Clang project, "Clang: a C language family frontend for LLVM," [Online]. Available: https://clang.llvm.org/index.html. [Accessed 12 September 2019].

[25] ArjoonSharma, "Codechef Competitive Programming - Problem statements and solutions provided by people on the codechef site," Kaggle, 2016. [Online]. Available: https://www.kaggle.com/arjoonn/codechef-competitive-programming. [Accessed 24 09 2019].