ECM3401 - Final Report
Developing a Probabilistic Model of Python Code

25th April 2019

Abstract

This project attempts to develop a probabilistic model of python code to allow for the generation of python code to prove four hypotheses. (1)The model can produce syntactically correct code, (2)the model can produce code that can run without runtime errors, (3) the model can produce code that can create an output and (4)the model can produce code that can create an output that changes depending on the input. Following a literature review of the current attempts to this problem, the n-gram model was chosen as the method to implement the probabilistic model. The n-gram model was trained using python data in JSON AST tree format, and subsequently, was implemented over a tree structure using a given node's parent, last sibling and position relative to its siblings to form the bigram. Other methods including variable renaming, a context function and function renaming were used to improve the quality of the generated code and reduce runtime errors. The n-gram model was trained, python code was generated in JSON AST format that was translated to python code, which was then executed using different global variable values to investigate if the model could prove the hypotheses. Nine experiments were carried out, each using a different corpus size (10,100,100) and variable maximum value (10,100, unlimited). The model proved hypotheses (1) and (2) achieving no syntax errors in 88% of cases and no runtime errors in 45% of cases. Hypotheses (3) and (4) were not proven, with future work suggesting an improved context function.

Contents

1 Introduction

There are many application areas in computer science for a probabilistic model that can automatically generate code. This project aims to create a probabilistic model, based on the use of n-grams, that is able to automatically generate python code, which could be used in a variety of ways. The project investigates the effectiveness of the implemented n-gram model at producing code based on four success criteria. (1) Is the code syntactically correct, (2) able to run without runtime errors, (3) able to produce an output when run and (4) able to produce an output that depends on a given input. The approach will train the model on a large python code corpus in an attempt to generate human-like code, that is code that looks as though it was written by a human.

Whilst there are many applications of automatic code generation, the three main areas that it would improve are software development, genetic programming and compiler fuzzing. For the area of software development there are many specific areas of use. IDE add-ons would provide the user useful tools for auto-completing or correcting their code whilst developing. Source code mining uses large repositories to identify common code, which can be automatically suggested in places where it is usually seen. Another more obvious use is the generation of entire programs that can solve a specific problem. All of these implementations of the model would save developer time, and the code generated code potentially be more efficient or a better implementation once the model is highly refined.

Compiler testing uses a process called fuzzing. This is where programs are randomly generated to test a compiler for bugs. Fuzzers require extensive development, are language specific and even the best fuzzers do not cover all program space. Also, the code generated by a fuzzer is not representational of a real-world example. This means that bugs will inevitably get through testing. By using a probabilistic model to generate the programs which test the compiler, these issues can be resolved. Development times will be shortened as the model can be used on multiple languages by just changing the training data, and more program space that represents real-world examples will be covered by learning from and mimicking actual program code.

Genetic programming uses the principle of natural selection to change programs over generations, until they reach a certain fitness as decided by a fitness function. It has been suggested that using a probabilistic model to generate the new programs in each generation allows for more effective mixing capabilities, than when using a standard genetic algorithm method with mutation and crossover. [19]. This results in fewer generations needed for a program in the population to reach the required fitness to be a success.

Firstly, a literature review and specification overview will be presented. This will outline the literature behind the topic, discussing current attempts at the problem of generating code and why the n-gram model was chosen. Following this, this document will contain the design, implementation, testing, experiments and evaluation of the project. The design section will present the chosen design at a high level. The experiment design will explain the different ways that the model will be evaluated, and which corpus will be used to do so. The development log will show the stages undertaken to reach the final product, discussing problems that were encountered along the way and how they were rectified. The testing section is primarily focused on testing and proving that the implementation of the model is correct, rather than proving that the model itself is useful. It is important to make this distinction so that the experiment results can be used to prove the successfulness of the model, knowing that the code implementation is correct and is in no way affecting the outcome. The experiments section will display the experiment results and analyse them, attempting to draw conclusions about the model used and its effectiveness at automatically generating program code. The evaluation will look at the project as a whole, seeing if the main goals have been achieved and if the project was approached well. This will be followed by any future work suggestions that could be carried out off the back of this project.

2 Literature Review and Project Summary

2.1 Literature Review

The literature review analysed a number of sources, that discussed multiple possible approaches to the project, in an attempt to deduce the most suitable approach for the project. This section will summarise the findings of the review, outlining the analysis that took place, which helped decide the method that would be used. The three main methods discussed are (1) probabilistic context free grammars, (2) the n-gram model and (3) neural networks.

Common across the methods in the literature being reviewed is the use of abstract syntax trees (AST) to represent the code structure produced by the model [15] [13]. Abstract syntax trees are a tree representation of the source code. Within an AST, each of the nodes represent a structure within the source code, such as a for loop. [8] The AST will not, however, represent every detail of the code. Since it is abstract it does not contain information on non-essential punctuation. They are used in compilers to represent the source code during the code parsing stage, so are highly suitable for the problem [12].

Probabilistic context free grammars (PCFGs) can be used to model the python syntax rules, which can be used to generate code. Context free grammars (CFGs) represent a natural language's syntax as a set of rules which define all the possible strings that can be created by a language. PCFGs assign a probability to each possible string based on a probability distribution that is learned from training data [7]. A generative model called PHOG (probabilistic higher order grammar) uses the basic methods of a PCFG but introduces the use of a context function to add more meaning to any generated code [4]. This method traverses the abstract syntax tree of already generated code to obtain information about the current context, using this information to alter the probabilities of potential generations. A similar method is also presented by Raychev et al [17] where the abstract syntax tree of the already generated code is traversed to calculate a context.

Neural networks are a series of neurons formed together in a network, inspired by the biological structures found in animal brains, that can learn how to do a task with supervised learning [18] [5]. Recurrent neural networks (RNN) have been implemented by Karpathy et al [9] to generate text including source code. An RNN is able to remember the previous executions of the network, forming a history of executions, by using recurrent connections in the hidden layer that allows the model to be more complex than a standard network [9] [14]. Deep learning is an alterative to RNNs, where models have been introduced [3] [6] that attempt to improve on current models of source code. A specific example is the DeepCoder implementation of deep learning that treats code generation as a search problem. The network searches across the space of all possible programs trying to find a solution to the problem.

The n-grams model defines a probability distribution that, given the previous n-1 items in the sequence, can predict the next item [2] [11]. The probability of a token, given the previous token in the sequence, is equal to their bigram, divided by the probability of the previous token. The N-grams model has been used successfully in the generation of natural language, providing many benefits over using a symbolic knowledge that relies on the grammar rules of the language being generated. The use of the N-gram model by Langkilde et al [10] provided many improvements to a symbolic model, which relied on alternative methods to work around decisions that it could not make. The introduction of the n-grams model improved the quality of the language generated and reduced the complexity of constructing the knowledge base. The statistical n-gram model is trained on a large corpus and can then identify the possible sentences from the word lattice generated by the symbolic generator. This example helps illustrate how the n-grams model can introduce meaning into the generated text so that the sentence makes sense, without the need for constraints to be placed on the model.

It has been shown that the n-grams model can produce the highest accuracy of predictions, surpassing neural networks and PCFG models, when the correct adaptations are made [6]. This, in addition to the complexity of creating and training a neural network model, make the n-gram model the best choice for this project. As seen from many different sources, ASTs are very effective for modelling program code, and subsequently will also be used alongside the n-gram model.

2.2 Project Specification

The following specification presents the requirements for the program. The n-gram model will be used to implement the probabilistic model and generate the python code. In addition, a context function, that will alter generation probabilities based on the code surrounding the token to be generated, will be used alongside the standard n-grams model. The n-grams model with be trained on a large python code corpus. The code input into the training will be in AST form for ease of use. The model will then generate the python ASTs, using the context function to improve the code quality. The ASTs generated by the model will then be converted back to python code so that they are in an executable state. The generated code is then evaluated based on the four criteria of success.

2.2.1 Requirements

1. Accept as an input a large python code corpus for training data
2. Train the n-gram model on the input data
3. Generate n abstract syntax trees using the n gram model
4. The AST generation must use a context function to alter the probability of possible generations
5. Generate python code from the python ASTs
6. Run the python code using the appropriate input data for the program, this can be in the form of setting global variables
7. Report output from the code execution, this may be an actual output or error messages
8. Collate results into success rates and data about program execution
9. Feasible performance is required to allow for testing in reasonable time
10. The program must have saveable state, so the program does not have to be retrained after execution

2.2.2 Hyptothesis and Evaluation

For this project there are four research hypotheses focusing on the probabilistic model. These will establish how functional the code generated by the model is.

1. The model is able to produce syntactically correct code
2. The model can produce code that can run without runtime errors
3. The model can produce code that can create an output
4. The model can produce code that can create an output that changes depending on the input

The hypotheses for this project will be evaluated by running the generated python code within another python environment, allowing for the control of any inputs and global variables programmatically. This allows for multiple rapid executions with different inputs and the recording of the execution results.

3 Design

3.1 Program Design

3.1.1 Overall Design

The program will be implemented as a single python file, responsible for all aspects of the project except for the collation of results which will be completed manually. Firstly, the n-gram probabilistic model will be trained on the python ASTs, which are in JSON format. Once the model is trained on the training data, then the probability distribution will be used to generate the program code. This will be used alongside a context function that focuses on variable types, to make sure that any variables being used are of the correct type for their context i.e. the operation they are being used in. The code generated in JSON format will then be translated into python code, before being run and tested against the research hypotheses. This section describes in detail each section of the program described above, together with the dataset and experiment design.

3.1.2 Probabilistic Model

The probabilistic model is implemented using n-grams. The traditional use of the n-grams model is on plain text, where the probability of a token is dependent on the previous token or tokens in the sequence. However, for this project the n-gram model has to be implemented on a python abstract syntax tree, as the training data and the code generated by the model is in AST form. This poses a challenge, as a decision must be made as to what counts as the previous token in the sequence and what forms the bigram.

The chosen implementation for the n-gram model over the AST was to count the number of times each element occurred, where they had the same parent, position with respect to their sibling nodes and previous sibling. This would form what would be the bigram. Therefore, to calculate the probability of an element in the tree, the total number of occurrences of the element with the current parent, sibling position and previous sibling would be divided by the total number of occurrences of that element in any situation. These elements are being used in the bigram, as in the python AST there are three important pieces of information that describe an element for the n-gram model, besides what the element is itself. (1) The parent of the element, (2) the previous sibling of the element and (3) the position of the element e.g. whether it is the first or second child. Firstly, the parent of the element is important, as within the python AST syntax a parent must have children with certain types, for example an if statement must have a body for the code to be valid. Secondly, the element that is the previous sibling is important as this aspect of the model is the most similar to the standard n-gram model, where the probability of the element depends on the n previous elements in the list. Here we are looking at the element directly before, following the standard roles of the n-gram model. Finally, the position or order of the children elements is important as some statements would become syntactically incorrect if the child nodes were reversed. This is true for the code x = method(), as method() = x is syntactically incorrect.

3.1.3 Model Training

The n-gram model will be trained on each python file by traversing the AST, which is in JSON format, in a depth first manner. For each node that it traverses it will check if the node has been seen in this context before. If it has, then the counter for the node in this context, i.e. its bigram, will be increased, indicating that it has been seen again. Otherwise, a counter for this bigram will be created in the model, indicating that this is the first time this has been seen. The term context in this case means if the node has been seen with this parent, position relative to siblings and previous sibling before. This training will store all occurrences of all nodes in each of their contexts and count the number of times that this occurs.

Variable renaming is an important part of the model training. Its aim is to try and cause code interaction between code sections from different programs that are in the generated code. When a variable is encountered in the training data it will be renamed before being placed into the model. Depending on the limit on the number of variables that are in the model, it may be renamed to be the same as a different variable in the same program. The result of this is having variables from different programs having the same names in the model. By standardising the variable names across all of the programs in the training data, the chance of a variable being used multiple times in a single generated program is highly likely, as the number of variables we have to choose from is much lower. Whereas before when the number of different variable names within the model is extremely high, the chances of seeing a variable even twice within the code is low. It also allows sections of code that have been generated from different programs to interact as they now have the same variable names. The same will also be done for classes and functions in attempt to allow for code that can interact with itself more.

3.1.4 Generation of Code

The code will be generated as a python AST in JSON format. For every iteration that a generation takes place, all nodes that could occur at this point in the tree are collated, with the entire model being searched for possible nodes. For each node that is found to be a candidate, the probability of the node is also calculated. This is done by taking the total number of occurrences of the node in the current context, divided by the total number of occurrences of the node in all contexts. Based on the probability distribution across all of the candidate nodes, one is chosen to be the node generated for the code. When variables are being generated, the context function ensures that the variable type is appropriate. More is said about this in the next subsection. To align with the renaming of function and class declarations in the training of the model, any method calls will also be renamed to match the names of functions that are available in the current code. This is important as without this the chances of generating a method call for a function that exists is low.

3.1.5 Context Function

The goal of the context function is to have the generation of an element be based on some of the code that has already been generated, so that the code is more likely to follow a path that can perform a useful calculation, rather than being random. The basis of the context function is to generate code, where the variables generated are of a type that makes sense in the current state of the code. Since variables across programs may have the same name but be a different type, there is the chance that a variable declared in the generated code is used in two places, but a different type is required in each for the code to be syntactically correct. This will cause issues when trying to globally declare variables, as discussed in the Translation of Code to AST section, as runtime errors due to a type mismatch could occur. The context function will therefore be responsible for ensuring that any variables generated are only used in places requiring the same type in the code, unless the variable is redeclared part way through the code. Not only will this remove the issue of runtime errors occurring, but it is also designed to improve the usefulness of the code and make it more likely to pass the hypothesis tests for this project. By using variables of a type that match their context, they are more likely to allow the program to do something useful with them, rather than operations that dont logically to the variable even if it is syntactically correct.

Each variable used will be stored along with its type. When the code is generated and a variable is required only variables that match the correct type constraints will be considered, altering the probabilities of those that are not applicable to zero. At any point an assignment statement is generated the variable will have its type updated to reflect what it can now be used for in the code. The main focus will be on restricting the interactions between integer, string literals and other variable types.

3.1.6 Translation of Code to AST

Since the dataset is in the form of JSON representing the python ASTs, it makes sense for the model to be trained on the JSON and then to generate the code in JSON format. The JSON format has much simpler rules to follow and it is easier to store the elements in the n-gram model as the language is less verbose. However, the code generated will have to be converted back to a python AST before being converted back to actual, readable python code. The python AST will be converted back to python code using the astor library [1].

After research that found a python library capable of translating JSON into a python AST is not available, it was decided that the code to translate from the JSON to the AST would have to be implemented. This implementation must handle any possible python structure that is seen within the test data, as if it is in the test data it is possible that it can occur in programs generated by the model.

3.1.7 Running the Program Code

The code itself will be run using the inbuilt python exec method. Each program that is generated can be run any number of times. The program inputs will be in the form of setting global variable values for any variables where it is required in the program code. Whilst the practice of setting global variables outside of the code may not follow good design practice in standard programming, it is important for this project as it allows for the investigation of inputs effecting the program output and if there are any diverging sections in the code, or if it is going to always output the same values meaning the code is less useful. Setting any variables as global also removes the issue of runtime errors caused by variables being used before they are declared. It is not a guarantee that the code generated will only use variables that have been declared by the code, by declaring them beforehand we have control of guaranteeing that this will happen in a simple way.

3.2 Dataset Design

It is very important that the data used to train the model is appropriate, as it is ultimately responsible for the code that is generated by the model. The dataset chosen to be used is the 150k Python Dataset [16]. This dataset contains 150 thousand python programs parsed into ASTs in JSON format. The files in the dataset were obtained from GitHub repositories with licencing that made programs free to use. The maximum length of any programs used is 30,000 AST nodes and an attempt was made by Raychev et al [16] to remove any obfuscated files. The dataset also contains no duplicated files. For the purpose of this project 150 thousand files is too many to use when training the model. Firstly, training the model and then generating code using it would take an unfeasible amount of time. Depending on the amount of time the program takes to execute when implemented the number of files used for training will be altered. Secondly, having a larger number of files for the model to train from will increase the different possible fragments of programs that the model can use to generate from. This will increase the chaos within the generations, making it less probable that the code will do anything useful, or that it will make sense. Increasing the number of files in training does allow for a greater number of possible programs to be generated, and as the number of possible programs increases the number of good programs increases. But for this project, where the training programs are very different and not trying to solve similar problems, it makes more sense to restrict the overall number, so the model does in some cases produce something good.

Although most of the data cleaning has already been done for this dataset, more is required for the data to be useful to this project. Due to the number of files being used in development testing and experiments, the data cleaning will be done manually. Any files that contain very small amounts of code, for example a comment followed by a single line class declaration, will be omitted from the training

Test Number	Corpus Size	Number of Variables	Criteria Met?(e.g. YYYY)
1	10	10	
2	10	100	
3	10	Unlimited	
4	100	10	
5	100	100	
6	100	Unlimited	
7	1000	10	
8	1000	100	
9	1000	Unlimited	

Figure 1: Experiment Plan

data. Any files that have code which could cause generated code that isnt useful, for example a class with many return functions for attributes, will also be omitted. Comments in code, whilst useless, will still be used in training as they do not alter the overall output of the programs.

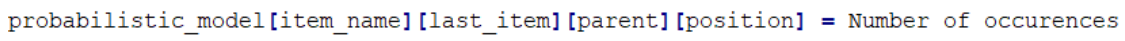
3.3 Experiment Design

The aim of the projects experiments is to see how effective the model is at generating python programs and the effect that chaning the corpus size and variable maximum has. Across the experiments the number of files in the corpus and the number of variables in each of the files will be changed.

As the number of files changes so does the complexity of the model. When the model has only a few files to learn from there is less code that it can splice together to form the final program. This will affect the end results as the more code you have to choose from, the greater number of possibilities there are of code generations. As more and more code is spliced together from different programs that do different things the chance of this code being able to do something useful decreases.

By restricting the number of variables that can be in the generated program it is easier to guarantee that the program will have no syntax errors. It will also aid in making sure that the variables are used in more than one place in the code. When the limit to variables is unlimited, if there are thousands of possible variables to choose from the chances of any variable data being processed beyond one calculation is low, as the variable is unlikely to be generated again. Variable restriction can therefore make the code more useful and actually increase the chance of it passing the hypothesis questions for the project. For each of the experiments 10 programs will be generated. Each program will then be tested against the research hypothesis. (1) Does the program run without syntax errors? (2) Does the program run without runtime errors? (3) Does the program produce an output? (4) Does the program produce an output that depends on the input? For each of the experiments the values of the variables declared globally will be changed and the code run for each of the different values. This will allow us to identify if the inputs to the program are causing a different output and therefore test hypothesis (4). Each experiment will also be repeated three times to eliminate any outliers from the test results.

In the project specification, one of the evaluation tests was to get the model to try and predict a specific omitted node from a section of code. This test is to be removed from the project experiments as it is not suitable for the type of model being used. The n-gram model to be implemented should always generate code that is syntactically correct as it is generating the code from the model which effectively stores the language syntax. Trying to test the model to predict python syntax would therefore be pointless as it would always be able to. The model is also trained on a large corpus. Getting the



```
probabilistic_model[item_name][last_item][parent][position] = Number of occurences
```

Figure 2: Data Structure of Probabilistic Model

model to try and predict something contextual, such as a variable or the next type of statement to be in the code, could have multiple correct answers as it may have seen multiple correct examples in the training corpus. There is therefore no single correct answer to this question. For this reason this test is not useful for this model and will not be included.

4 Development Log

This section will describe in detail the development process of this project, split into subsections to describe each individual stage. The stages that formed the development process were: producing the prototype model, improving on major issues in the program, adding the final stages so the program can be used from generation all the way through to testing the generated code and then improving the model to produce better and more useful code based on the testing metrics.

4.1 Producing Prototype Model

4.1.1 Implementing Basic Model Training

The first task for development was to train the n-gram probabilistic model on the training data. This would involve storing the number of occurrences of each bigram, for every element in the training data. The main problem with training the model was trying to find a way to store the information obtained from the training data within the model, whilst being logically correct and efficient. For each different element found within the code, there would be many possible bigrams that need to be stored. For a standard n-gram model, perhaps the most sensible data structure to be used would be to have a dictionary for each element that would store the other element in the bigram as the key, and the number of times it has occurred as the value. However, this method cannot be used in this case, as there is more than one element to each bigram, being the parent, position and previous item of the element. It would be possible to have a dictionary entry for each bigram for the given element, where the key would contain some form of concatenation of the parent, last item and position as to where it could be identified uniquely. This would be extremely inefficient when looking for possible candidates when generating code however, as each element would have to be inspected to see if it is viable. Another solution is therefore required as the standard model for storing n-grams does not fit.

An alternative would therefore be to have a nested dictionary structure. Each item x would have a dictionary where the key was the last item e and the value would be another dictionary. This dictionary would then have the key of all the parents p, where x has the last item e. The value would be another dictionary. This dictionary's key would be all the positions n where the element x has the last item e and the parent p, with the value being the number of times this combination of x,e,p and n has occurred in the training code. The data structure for the model can be seen in Figure 2. This method of storing the model data is much more efficient when it comes to generating the program code. When searching for a suitable candidate, if the last item it is looking for is not found it can immediately discount this dictionary and any nested dictionaries for that item, removing the need for many pointless comparisons when trying to find a candidate. For the initial prototype implementation, the position element n of the model was not included to simplify the model structure, making it easier to test.

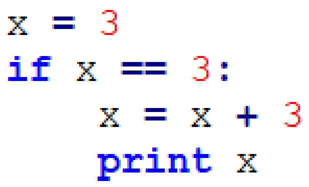
The training data was in JSON format, with each element representing a node of the python AST. The format of the nodes depended on if it was a parent or a leaf node. A parent would store the type followed by the list indexes that referenced the child nodes. A leaf node would store the type of node followed by its value. The two different types of nodes would therefore have to be processed differently. This caused some inconsistencies in what was being stored in the model, as with the parent nodes the type was being used, e.g. If for an if statement, and for leaf nodes the value, e.g. x for the name of the variable. For the model to be useable either it would have to differentiate between what it was using, treating the two nodes differently at later points in the program, or the names and values of the leaf nodes could be lost, with the types of the nodes being stored instead. At this point it important to focus on what the project is trying to achieve and the methodologies that it is trying to use. The n-gram model should focus more on what the node is literally, rather than its type. Although we are using types with the parent nodes, the types match exactly what is in the written code, for instance the if type is still if within the program code. The values of the nodes should therefore be retained to keep as much contextual information and meaning as possible from the code, even if it does require more handling of the types of nodes in the model.

4.1.2 Generating Basic Code from Prototype Model

At first, to keep the model simple, the code output from the generation stage was in a basic tree format, where each parent pointed to child nodes using an index list and any leaf nodes had only a value and not a type. Initially the code generation was performed by one loop, looking for candidates and then assigning probabilities based on the total number of times the candidate is in that context, divided by the total number of times the candidate has been seen. Initial testing proved however that this method would not work. The model had no way of distinguishing if a node was going to be a parent by probability and could not correctly add child nodes to the parents, leaving empty structures such as blank print statements. A new method was introduced that would generate the code in depth first order. A node would be generated then an attempt would be made to create a child node, if successful the process would be repeated. If not, then all siblings of the node would be created. This method would ensure that the entire depth of the tree had been fully explored and that no nodes in the tree would have missing children nodes.

4.1.3 Testing Prototype Model

The model at this stage of development was very limited, so the tests were aimed at proving the code implemented so far works with very basic tests. The aim was to provide the model with a single python program, with only a few lines of code and see if it was able to reproduce the code in the generation phase. An example of this program can be seen in Figure 3. By using the tool provided by Raychev et al [16], a small section of handwritten code was translated into the JSON format. After minor bug fixes the model was able to train from the code successfully. It was difficult to guarantee that the model training was 100% correct, as reading the nested dictionary structure was challenging. This is why for this initial test; the training corpus was kept small. Whilst the model was able to successfully train, the code generation had issues. Since the location information was yet to be included in the n-gram model, on some occasions the programs generated would be extremely long. In some cases, the model continually generated code until it had to be manually stopped. This highlighted the need for a manual hard coded limit to the number of child nodes a single parent could have. Perhaps a more elegant solution would be to train the model in such a way where it would do this following the standard logic, rather than including a hard coded rule. This would mean that the model is actually learning the desired behaviour organically, rather than it being forced behind the scenes. In terms of the design principles of the project it is always better to achieve the project goals using the model as much as possible. A program more model reliant will be more resilient and adaptable to handling different scenarios, as well as generating code that is more diverse and based on the original corpus. Reading the outputs from testing at this stage was difficult due to the loose JSON format that the data was being presented in.



```
x = 3
if x == 3:
    x = x + 3
    print x
```

Figure 3: Example of a Simple Program Used in Testing

Going forward it would important to implement the translation to readable python code, as this will allow the evaluation of programs to be much easier.

4.2 Improving on Major Issues

Following the initial testing of the very basic prototype program, many major issues had been identified in the program. The goal of the next stage of development was to fix existing issues, whilst adding new code to the program that would improve the functionality to make testing easier.

4.2.1 Changing Logic to Fix Program Length

Before adding any additional functionality into the code, the major issue of the program generation never stopping had to be addressed. Another issue that was later thought of, but not observed in the prototype training, was an incorrect number of nodes being assigned to a parent. In the AST syntax, some nodes must have a certain number of child nodes e.g. a comparison operator can only operate on two nodes. Since the model was not implemented with logic to learn the possible number of nodes for a parent, if the program generation did produce something in previous testing it would be syntactically incorrect.

There were two factors that contributed to the code generation never stopping or not assigning the correct number of nodes to a parent. Firstly, the code generator cannot calculate the probability of there not being a node after the node it is currently generating, meaning that the generation will only stop if there are no candidate nodes that can it can choose from. A loop is possible in the model that will allow it to have infinite candidates. This occurs when two nodes have the same parent, and both follow each other at some point in the training data. For instance, if x + y is followed by y + x. Both x and y share the same parent, being the Add node, and in the AST they both have each other as a last item. When generating the code the generator will have no probability of when to stop, so the infinite generation x + y + x + y + x .. will occur. To fix this issue, the model was altered to store an end of line delimiter once it had reached the end of a set of siblings. This delimiter is stored as though it is a standard node, meaning that it will have its own parent, last item and position elements in the model. When the model is generating the code it now has a chance to generate this delimiter. If it does, then the set of sibling nodes is complete, instructing the model to no longer try and generate child nodes for the parent. The model is therefore able to calculate the probability that there are no more nodes, including it in the generation.

Secondly, the model was not learning about the position, or appropriate number of nodes to assign to the parent. The plan of implementing the position of the node as part of the bigram should fix this issue. The initial design of implementing the position of the node into the model was originally thought of to enable the model to learn the correct order of the child nodes. Whilst this is partially achieved by including the last item in the model, by including location in the model as well this enables the n-gram model to have longer range dependency, almost as though it is using a value of n greater than two as we are for this adaptation of the bigram model. By including the position as part of the learning the model would know the limit to the number of child nodes for any given parent node. For example, in the training data there would never be a case where a comparison operator would have three child nodes. Therefore, in the model, there would never be a node that had a position value of three, whilst having a comparison operator as its parent. The model would therefore be unable to generate any code where a comparison operator had three child nodes, fixing the issue.

Whilst including the position as part of the model does appear to fix both issues, there are some cases where the model having the ability to include the probability of no more nodes causes different outcomes. If we have two lines in the training program code, x + 1 and x + 1 + 1 , the line x + 1 will never be generated. Since the model would have the option to choose another 1 as a child node it would, as there are no other choices contesting it. By including the chance of no nodes using the delimiter logic however, the model can choose to generate no more code, allowing just x + 1 to be generated.

4.2.2 Implementing New Functionality into the Model

The second part of this stage of development was aimed at improving the functionality of the model, not necessarily to improve the generated code, but to improve the usability of the model, making it easier to test. The first task was to add the functionality of training from multiple files. In terms of adding the code into the model it was a very simple change. The training process had to be repeated for each of the files in the training set, retaining the information that was learned during each iteration. The more interesting part was implementing a way to decide which files from the training data should be used in testing. A program that selected at random a specified number of files from the training corpus was implemented, with the goal of choosing the files for training and testing the code. Since the program was choosing the files at random, many had to be discarded and replaced as they were not suitable for training. This was due to their very short length or contents that were deemed not useful.

During this stage, the generated code was also improved so that it was in the same JSON format as the input data. Since only the values from the training data are being stored in the model, information is being lost about the nodes. This means that when the program code is generated, the type of any child node is not known. A simple workaround was implemented that would store the type of any child node in the model, matching the name of the node to its type in a dictionary. Whilst this solves the issue and allows the code to be generated in JSON format, the type that is assigned to the final node may not be correct. If two variables have the same name in the training data but a different type, only one of the types is stored against the same name. This could result in a node having a different type to what it had in the training data. Since there is no way to differentiate between nodes of the same name once they are in the model, it is taken as a limitation of the model that a nodes type may change. Due to python not having strong types, the chance of this happening is very reduced as there are less types that a variable can be. Also, the variable will only be changed to a type that has been seen in the training data, so the result is still a valid candidate and not something completely random.

4.2.3 Testing the Major Code Improvements

At this stage of development, the code corpus is kept very basic so that the code correctness can be focused on, rather than the ability of the model to produce useful code. The tests were also restricted to one variable, as the issue of variables causing TypeErrors and being used before declaration had yet to be addressed. Five python programs of up to six lines long, containing only assignment, print and if statements comprised the test data. Multiple tests were run against the model. The programs being generated were now no longer than any found in the test corpus. There were also instances of programs being generated that were shorter than those found in the corpus. This confirms that the logic implemented in this stage to remove generation loops and the logic to allow for the calculation of the probability of no more nodes was successful. The statements from the different programs were being combined successfully and the programs produced didnt look like copies of any of the original programs. This confirms that whilst the number of variables is kept at one, the model is able to splice together sections from different programs in the corpus, something that is very fundamental to the success of this project. Whilst no issues were observed that could have been caused by the types of nodes changing from training to generation, this is a problem that may have impacts when the code corpus and generated programs become much more complex later in the project.

4.3 Completing the Model from Training to Code

At this stage in the project the prototype version was only implemented up until the generation of the JSON code; still missing was the translation from the JSON to python code and the executing of the programs multiple times using different values for testing. The aim of this stage was to complete the model, so that it can be trained on the python corpus and generate python code that can be executed and tested.

4.3.1 Converting the JSON Output to Python Code

The first task was to implement the translation of the JSON code to the python AST. Some research was done to see if there was an already existing library that translates code from JSON to the python AST. Unfortunately, nothing could be found, so an initial implementation was attempted, which was able to translate the JSON from the current tests into the python AST. This implementation was however was specific and could not handle many python structures at all. The way in which the python AST worked made it very difficult to implement any generic functions that could handle different types of nodes. After some investigation, it was discovered that for the implementation to work for any python code, the translator would have to process every item in the python grammar differently. The standard python AST library is very limited in functionality and usability. For a node to be created, the type of node and all of its attributes have to be hard coded. The documentation for the library is also very limited. The grammar is shown but there are no descriptions of what each of the nodes or their attributes mean. This makes it very difficult to relate between the JSON code and its equivalent AST representation.

To make the process easier and to understand what the AST attributes relate to, the original python code to JSON AST translator provided by Raychev et al [16] was used. By reverse engineering the code some sense could be made of what the JSON represents in the AST. However, when the python code is translated into the JSON AST some information from the code is lost. The less important attributes are not translated across as they are not vital when using the JSON code in the model, and some nodes are treated generically causing more attributes to be lost. Many nodes are also renamed to make them easier to use in the model e.g. the add operation is changed from the type BinOp with the Add attribute to the type BinOpAdd without the attribute in the JSON code. This made a direct reverse engineer of the code impossible, as the AST module does not allow for the generic nodes to be created.

Since it was becoming clear that the implementation of the JSON AST to python AST translator was going to be difficult, an alternative method was investigated. Instead of using the code in JSON form, the python AST could be easily obtained from the raw code. If the model was trained on the AST nodes and then the code generated using the nodes no translation would be required. It was found however that this would not help with the issue and would instead just move the problem to other areas of the model. The overriding issue is that when processing the ASTs the code doing so has to be very specific. Each node may require something different and when creating a new node; the type has to be hard coded. Introducing python ASTs to all areas of the model would make it significantly more complex, increasing the likelihood that the code correctness would interfere with the model output. Raychev et al [16] have decided to translate the python AST into JSON ASTs for a reason; that they are much easier to manipulate and use. It was therefore decided that the JSON ASTs should be used and the translation from JSON to AST would have to be implemented. This would take some time and was not useful in investigating the success of the n-gram model at producing code. It was necessary though, the code had to be executed as manually viewing and running through the output JSON with multiple variables would take an unfeasible amount of time.

4.3.2 Running the Generated Program Code

To execute the program code the exec() method is used on the code output from the Astor [1] library. This allows the program code to be executed programmatically, so that tests can be carried out in quick succession. The generated code is also output to a file so that the model's success can be viewed subjectively, based on how good the code looks.

At this stage in development, any variables that are in the generated program code are declared globally in the in the exec() method parameters. This was initially implemented as a quick way to ensure that the variables in the generated program would be declared before use, so that this problem would not cause runtime errors. The issue with declaring variables globally is that it is detracting away from the original project aim to generate the code using the n-gram model. The more the code is controlled manually the less the n-gram model is doing. When the project aim is to investigate the effectiveness of the model, reducing the effect of the model on the outcome reduces the usefulness of the results, as they show less and less about the model. However, in this case the problem with variables being used before declaration is very common in previous attempts found in literature and is difficult to solve. Without having programs that can execute, very little amounts of testing and experimentation can be done. By using the global variable workaround to remove this bottleneck, other useful investigation can continue which may produce useful results. The overall model may end up better due to less time being spent on this difficult issue.

4.3.3 Testing the Model

The JSON AST to python AST translator required extensive testing to ensure that all possible cases of python code in JSON AST form could be translated back into a python AST. To try and isolate the code, the translator was tested by taking programs from the code corpus, translating them into the JSON format, then attempting to translate them back into python code. The input program and output from the translator could then be compared; ideally they should be identical. This process was very extensive. Often, new programming structures would either cause the translator to fail, as it was unable to handle the code provided, or the Astor library could not translate the produced AST into python code. The Astor program would fail to translate the AST if it was not in the exact form required. The library was poor at handling any slight differences in the syntax and any error messages that it provided were very non-descriptive. This made the problems very difficult to debug as it was sometimes difficult to even find the node containing the error. If there was a single problem in the entire AST then no code would be produced. Often, the piece of code being translated would have to be converted into the AST and manually compared to the original to try and locate any differences. After many tests and additions to the code the translator appeared to handle any python program that was drawn from the corpus. It is difficult to say it the translator is able to handle any possible scenario, but for the purposes of this project it is highly likely to be sufficient. If any code translation errors occur during testing, minor alterations could be made to the code to fix the issue.

4.4 Improving the Model to Generate Better Code

Now that the model is able to train from the corpus, generate the code and execute the tests it is at a complete state. The goal of the final stage of development is to improve and enhance the model. This will be a large stage where most of the interesting work is done and the testing will become more complex, using more complicated programs. The current model follows a standard n-gram model very closely, except that this model uses the nodes parent, last item and position. New additions to the model will deviate from this in an attempt to improve the generated code so that it can prove the research hypothesis.

4.4.1 Increasing Test Corpus Size and Complexity

Up until this point in development the code being used for testing is very basic, being only a few lines long and using only one variable. Before enhancing the model any further, testing using more complicated data was carried out to see what needs improving within the model. One of the main issues found was the lack of interactions between different sections of code. This was to be addressed by renaming variables in the code. Variables were also being used in multiple places, causing conflicts where their types didnt match up. This was the reason that a context function was being used; to reduce the number of occurrences of this happening.

4.5 Renaming Variables

A key principle in achieving success in the model is having parts of code from different programs interacting to produce an overall different program. When the testing code corpus was changed to use more than one variable, statements were being generated from different programs and would work in the same python file. However, since the variables of the programs in the corpus had different names, statements from different programs were never operating on the same piece of data or interacting. Whilst this is less of an issue with a smaller corpus since there are less statements to choose from, as the chance of statements interacting is likely, with a larger corpus the chance of any two statements interacting is very slim. The result of this is that programs being generated are not useful and do not do much computation at all. To prevent this from happening the method of renaming variables was introduced. Renaming variables has an interesting effect on the code being generated. By standardising the names of the variables across the corpus, not only are variables likely to appear more than once in the generated code, but the fragments generated from different programs are likely to interact as they now all share variables.

There are two ways in which variable renaming could be implemented into the model. Either at the training stage or the code generation stage. When implemented at the training stage, the model will rename any variable that it comes across to a predefined value before learning from the program. The model will then generate the code using these renamed variables. When implemented at the generation stage, the model will train from the data as usual, generate the code and then post generation rename any variables in the code to a value in a set of predefined names. Whilst implementing the variable renaming at the generation stage allows for a simpler context function to be implemented, using variable renaming when learning from the corpus is the superior method. Since the generation stage renaming would be choosing from a set of variables, there is no way to have an unlimited variable count in the generated code. With renaming at training a maximum number of variables can be set, as some variables can be mapped to the same name, or it can rename all variables uniquely to be unrestricted. This method is more organic and requires less manual intervention; the generated code method is more reliant on the model itself rather than using restrictive manual changes. The code generated when variable renaming is at the training stage is also more likely to be interesting. The code is more likely to be carrying out related operations sequentially as the variable nodes in the model have more candidate nodes. Therefore, variable names which only appear a very few amount of times in the corpus with a small number of candidates will not halt the program generation early. The variable renaming was therefore implemented at the training stage.

4.5.1 Implementing Context Functions

As discussed before, the context function is responsible for ensuring that the variable generated has a type that is suitable for its context. Without a function to check that the model is generating a variable which has a type suitable for its position, there is no guarantee that runtime errors will not happen. The context function also serves the purpose of ensuring that the variable renaming does not cause type conflicts. When the number of variables in the model is restricted, variables from a program may share the same name. The two variables may have different types, meaning that they are often seen in different contexts. But, as they share the same name the variable may be generated in the two, completely different contexts that require different types, causing a runtime error.

The implementation chosen for the model was based on looking at the types of the siblings of any given node and seeing if they match. If a string literal is generated, then all siblings of that node are to also be string literals; this is achieved by making the probability of all non-string nodes zero. This is also repeated for any integer values. This implementation of the context function applied to types is very simple as the python language is not strongly typed. It is therefore impossible to tell during generation what the type of a variable will be since the JSON AST stores all variables under the same node type.

4.5.2 Altering the Body Node in the Training Data

Within the training data, any node that has a body of code, e.g. a function will have a function body containing all the code within that function, will have a body node as a child which will point to all the statements in the node's body. The body node that points to the code content is the same type no matter what the parent of the body node is. This means that when the model inspects the training data all body nodes will appear the same in the model. Therefore, during the generation stage of the program, code that was previously in one programming construct's body could now appear in another's e.g. a piece of code that was in an if statement is now generated in a for loop. Not only is this not technically following the model design, where code should only appear in the generation in places that it has been seen before, but it could lead to syntactically incorrect code and reduce the code's usefulness. Some syntax, for example in a class body, is specific to that node type. An __init__ function declaration only occurs within a class; without any changes to a model this could be generated in any body statement. Generating code in a body where it has been seen in the training data increases the chance of useful code being generated. For instance, it is more likely that code seen in a loop structure in the training data is suitable for a loop in generation as it is designed to be repeated.

To address this issue, the training data was altered before the model started training. Any body node that was in the training data was renamed to be body followed by the name of the body's parent e.g. bodyFunctionDef for the body of a function. Therefore, when the model trains on the data there will be multiple different types of body node in the model. By differentiating between the different types of body nodes in the training data, the model can only generate code in a body if the code has been seen in that type of body before.

4.5.3 Maximum Depth of Generated Code Tree

When using a very small training corpus less than ten programs in size, generated programs could become very long to the point where the generation function crashed due to the maximum recursion depth being reached in python. This matches the error that was seen earlier on in development, where there could be a loop leading to very long programs, but this time the error is caused by the generated AST becoming too deep instead of wide. Since there is nothing in the model relating to the depth of the AST, the generations are never limited because of their depth. Also, there will be no limiting by the end of line delimiter as a parent node that is not a leaf must always have a child node assigned to it. This problem mainly occurred when two Call nodes would have each other as child nodes recursively; in code form this would be a call statement nested in another call statement. In a small code corpus, the only time a call is used it may be nested with another call statement. Therefore, there is a chance that the AST depth will be far too high causing errors. Whilst this error is still possible in the model, it is only going to occur when the corpus is very small and the programs within the corpus simple. This means that in any of the experiments for this project this issue will not be able to impact the results, therefore no changes were made to the model related to probabilities of the AST depth. This is something however that should be changed in the future so that the model is more capable.

4.5.4 Renaming Generated Functions and Function Calls

One of the main issues identified in development testing was the inability of the model to use any functions that had been declared, or make function calls for functions that could later be generated. Since the model was not storing the names of the functions found in the training data, they were already being renamed to generic function names. When generating calls to functions however, the model was not using functions that the model itself could generate. This means that there would never be a case where a function generated by the model is used by calls generated by the model.

To solve this issue, any calls that the model generates are renamed to any function that the model has already generated, chosen at random from a list of the already generated functions. If no functions are yet to be generated, then the call is renamed to the first possible name that a function will renamed to. Not only does this model change allow for the function code it is generating to be run, but it also reduces the chance of runtime errors as the possibility of undeclared functions being called is dramatically reduced.

4.5.5 Optimising the Model

In the original specification, one of the requirements was the ability to save the probabilistic model, so that it did not have to be retrained for every execution. During development however it was quickly found that the training section of the model was very performant, it was in fact the generation section of the model that was causing the greatest number of performance issues. It was therefore decided that the ability to save the model state was unnecessary, and so the decision was made for it to not be implemented. Due to the structure of the saved model being a large nested dictionary, the model when generatinghas to iterate over the data structure many times. This is a very expensive operation and was targeted for performance improvements. One of the main changes to improve the model performance was to keep a running total of the number of times a node occurs whilst training the model. This means that the number of nodes did not have to be counted during generation for each node to work out its probability of generation, massively reducing the performance impact of generating the next node.

Since the model cannot differentiate between the types of nodes when generating, all string literals that are put into the model are renamed to contain a string delimiter. This was required as in some cases string literals in the corpus code were the same as the types of other nodes. For example, the string body was in the corpus code. When this was put into the model its data would be combined with that of the body node type. This would lead to the incorrect generation of nodes; the JSON AST to python AST translator could therefore not process the AST.

4.5.6 Assigning Variable Values for Testing

Initially, when choosing the global variable values for testing, the variables were assigned an integer value at random in the range of one to one-hundred. This was a very crude implementation however, as the variable in the training data may have never been an integer. Subsequently, the code where the variable is used in the training data and the code generated by the model may not run without error if the variable is an integer. If, for example, a string operation is generated and performed on a generated variable, the generated program when run will cause a runtime error.

A more organic solution was therefore devised, where the model is used to assign the value to the variable. The model is searched for a situation where the last item is the variable to be assigned to, the parent is the Assign node and the position is one. The result of this is looking through the model to try and find a node that has been assigned to the variable in the training data i.e. it is trying to find the value of ? in x = ? by looking at the model. By using the model to generate the value of the variable, not only are we likely to get a value that works in line with how it is used in the generated code, but the manual impact on the model is also limited. In the case of the search not finding an instance where the variable is assigned to in the model, the old solution of assigning the variable a random value in the range of one to one-hundred is used.

4.5.7 Final Model

The final model was able to train itself using a large python code corpus, generate the python code, translate the code to the python AST format and execute the code with multiple variables. The model stored the information it gathered from training in a nested dictionary structure. The use of a node's parent, last item and position in the JSON tree formed the components of its bigram. When generating code in a depth first order, the model uses variable and function renaming alongside a context function to generate the code. The aim of these additions is to reduce the chance of runtime errors in the code and to increase the probability of the code's 'usefulness'. Once the code has been generated, the model assigns global variable values before executing the program code. This is repeated multiple times to test the generated program against the research hypothesis. The model is now at a complete stage for this project and is ready for testing and experimentation.

5 Experimentation and Results Analysis

5.1 Code Testing

This section will cover the testing of the code to prove that the implementation is correct and working as intended. This is important to show that the results from the experiments are credible. By proving that the code is correct, the possibility of implementation errors affecting experiment results is removed, so that we know that only the model itself is affecting the results.

5.1.1 Model Training

The main aim of the training testing is to ensure that the data being captured during training is correctly stored in the data structure. Due to the complexity of the data structure storing the training data, the best way to test this section of the code is to present a small piece of code that covers all possible cases, or types of code that the model may see, so that the results can be manually checked. To cover all areas of the model training code, the test program should contain a statement with child nodes, a string literal, an integer value and a node without a value field e.g. a variable being assigned to an empty dictionary. The code should also be run with the variable renaming limit set to a value above and below the variable count in the test program, and an unlimited variable count. Following these test cases, the training section proved successful, correctly storing all data into the nested dictionary structure.

5.1.2 Code Generation

Testing the code generation stage of the model is challenging. The logic used to determine the node that will be generated is very complex, and when there is a large amount of data it is very difficult to test. The data used for this section of testing was therefore kept very small. Firstly, a single program with no repeated statements or variables was used. From this the model should, and did, produce the exact same program as that is the only possibility. The model was then tested with five very small programs that would be able to cover all cases in the logic of the code generation. For this to be achieved the test code contained function declarations, function calls, string and integer values used at least once in a statement and nested statements. The code generated from the test data was a combination of the test files and applied the rules of the context function correctly. Whilst this test does ensure complete code coverage and that the model is successfully generating code, it is still difficult to say that the code is being generated as intended. The validation only comes from manually looking at the code; there are no metrics to guarantee that the model is choosing the next node based on the probabilities we think it is.

5.1.3 Translation from JSON AST to Python AST

Much of the testing for this section of the code was completed during the development stage of the project. In order to implement the translation, programs were translated into JSON ASTs then directly translated into python ASTs, ignoring any training or program generation logic. This would identify any nodes that the translator was not able to process and allowed for the result to be compared to the original, so that the result could be checked. This testing that was used during development is also used to prove that the translator correctly processes the generated code. If the result of the translator was the same as the AST generated by the python AST library then the code is correct. A selection of 10 programs that has a minimum of 1000 nodes in the JSON AST representation were run through the translator. It was able to correctly translate the JSON AST back into the python AST, with only minor differences. The differences in the AST, such as the naming of functions, were a result of the code generated by the model being slightly different syntactically to the JSON ASTs of the training data. During some program generation, using the largest experiment corpus, the translator was unable to translate approximately five nodes that it encountered. To rectify this issue the code was added so that all nodes within the experiment data could be translated.

5.1.4 Executing the Generated Code

To test the execution section of the model, a simple python program was fed into the function, along with the list of all variables used that would need to be declared. Since the program would not be in the model the value of the variables would be set to a random integer in the range of 0 to 100. The program was therefore designed so that any integer values would not cause runtime errors. The program was successfully executed, and the values assigned, resulting in different outputs from the program each time.

5.2 Experiments

There are two parameters which will be changed in the experiments; the training corpus size used to train the probabilistic model and the maximum number of variables in the trained model. There are three corpus sizes which will be used, each will be tested using three maximum variable values. The generated code output from the model will be evaluated against the four criteria set out by the research hypothesis.

(1) Does the program run without syntax errors? (2) Does the program run without runtime errors? (3) Does the program produce an output? (4) Does the program produce an output that depends on the input? Each experiment will be repeated ten times and an average of the results will be taken. For hypothesis number (4) the criteria were slightly changed after some preliminary testing. Since an output would require a statement such as a print, or a file output, the chances of having an actual output from the program were very slim, especially because many of the files in the corpus would be a file containing just functions that works alongside the main file, meaning that it provides no code to the model that can generate an output. The definition of an output in our hypothesis is therefore changed to mean some kind of return statement or variable assignment that could be used as an output in some way.

5.2.1 Experiment Parameter Values

The experiment design is focused on finding out the effectiveness of the model when using a different sized corpus and variable limit. The expectation of the model is that as the variable limit increases, the programs will become more random and sporadic increasing the chance of runtime errors. As the corpus size increases, the variation between generated programs will increase.

To choose the parameters for each experiment the following considerations had to be made. Firstly, a feasible corpus size had to be used so that the program could run in a reasonable amount of time. The original corpus was 150 thousand python files in size. Using a corpus this big would make the experiment execution take a very long time. Also, beyond a certain point the model has so many programs to choose from that the chance of unique code being added to the corpus would be unlikely and would simply increase the amount of time that it would take to process. Secondly, on the other end of the scale, as long as the corpus size was more than a single file, interesting results could be drawn from the model. The model would still be able to generate different programs, although less varied, and it would be easier to see what the model was doing as the corpus could be inspected to see where the generated code originated from. The values of 10, 100 and 1000 were chosen for the corpus sizes. Beyond 1000 programs little change was noticed in the structure of the code being generated. A middle value of 100 kept the programs more directed to a specific task as less code could be spliced together, but still allowed for more varied generations than when the corpus was only 10 programs.

When choosing the variable limit for the experiments, it is important to choose values that allow the program to use the same variable in statements multiple times, whilst not limiting the programs functionality. A very low value of 10 allows for the model to manipulate the variables heavily, whilst not creating overly complex code. The increased value of 100 variables, especially as the corpus size increased, provided a good balance between variables being used multiple times, whilst being assigned and appearing in the same section of the code. The purpose of testing with unlimited variables is to see how the model behaves when there are no restrictions. At this point there is little chance of cohesion in-between statements, but ultimately having a model that is unrestricted in what it can do whilst working well is the ideal scenario.

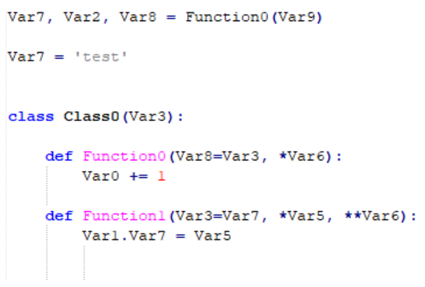
5.3 Results Analysis

5.3.1 Experiments 1-3

Each experiment results table shows the experiment number and whether it was successful in achieving all four research hypotheses. From experiments 1-3, presented in Figures 4-6, we can see some success in managing to successfully prove all 4 hypotheses. As the corpus size increases the overall effectiveness of the model reduces. Syntax errors started to occur at higher corpus sizes and runtime errors became more frequent.

Experiment	1	2	3	4
1	Y	Y	N	N
2	Y	Y	N	N
3	Y	Y	Y	Y
4	Y	N	N	N
5	Y	N	N	N
6	Y	Y	Y	N
7	Y	Y	N	N
8	Y	Y	Y	Y
9	Y	N	N	N
10	Y	Y	Y	Y
Total	10	7	4	3

(a) Table of results



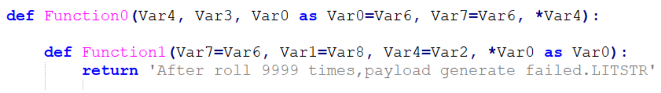
```
Var7, Var2, Var8 = Function0(Var9)

Var7 = 'test'


class Class0(Var3):

    def Function0(Var8=Var3, *Var6):
        Var0 += 1

    def Function1(Var3=Var7, *Var5, **Var6):
        Var1.Var7 = Var5
```

(b) Example code

Figure 4: Experiment 1 Results

Experiment	1	2	3	4
1	Y	N	N	N
2	N	N	N	N
3	Y	Y	N	N
4	Y	Y	Y	Y
5	Y	N	N	N
6	Y	Y	N	N
7	Y	N	N	N
8	Y	Y	N	N
9	Y	Y	N	N
10	N	N	N	N
Total	8	5	1	1

(a) Table of results



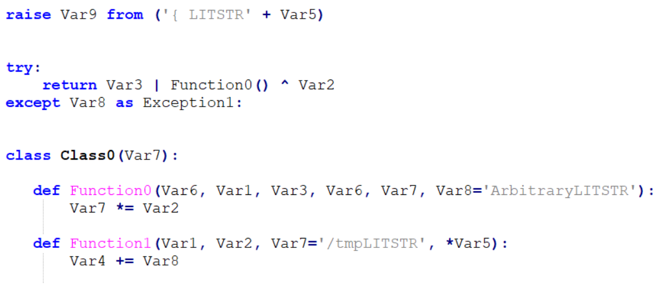
```
def Function0(Var4, Var3, Var0 as Var0=Var6, Var7=Var6, *Var4):

    def Function1(Var7=Var6, Var1=Var8, Var4=Var2, *Var0 as Var0):
        return 'After roll 9999 times,payload generate failed.LITSTR'
```

(b) Example code

Figure 5: Experiment 2 Results

Experiment	1	2	3	4
1	Y	N	N	N
2	Y	N	N	N
3	Y	N	N	N
4	N	N	N	N
5	Y	N	N	N
6	Y	N	N	N
7	Y	N	N	N
8	Y	Y	N	N
9	N	N	N	N
10	Y	N	N	N
Total	8	1	0	0

(a) Table of results



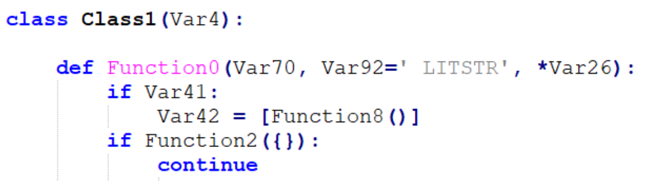
```
raise Var9 from ('{ LITSTR' + Var5)


try:
    return Var3 | Function0() ^ Var2
except Var8 as Exception1:


class Class0(Var7):

   def Function0(Var6, Var1, Var3, Var6, Var7, Var8='ArbitraryLITSTR'):
       Var7 *= Var2

   def Function1(Var1, Var2, Var7='/tmpLITSTR', *Var5):
       Var4 += Var8
```

(b) Example code

Figure 6: Experiment 3 Results

Experiment	1	2	3	4
1	Y	N	N	N
2	Y	Y	Y	N
3	Y	Y	Y	Y
4	Y	Y	N	N
5	Y	Y	N	N
6	Y	N	N	N
7	Y	Y	N	N
8	N	N	N	N
9	Y	N	N	N
10	Y	Y	N	N
Total	9	6	2	1

(a) Table of results



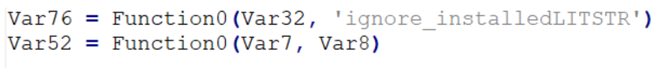
```
class Class1(Var4):

    def Function0(Var70, Var92=' LITSTR', *Var26):
        if Var41:
            Var42 = [Function8()]
        if Function2({}):
            continue
```

(b) Example code

Figure 7: Experiment 4 Results

Experiment	1	2	3	4
1	Y	N	N	N
2	Y	Y	N	N
3	Y	N	N	N
4	N	N	N	N
5	Y	Y	Y	N
6	Y	N	N	N
7	Y	N	N	N
8	Y	N	N	N
9	Y	Y	N	N
10	Y	Y	N	N
Total	9	4	1	0

(a) Table of results



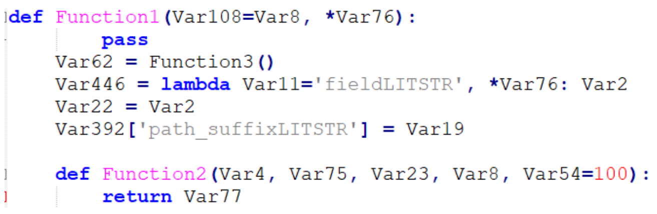
```
Var76 = Function0(Var32, 'ignore_installedLITSTR')
Var52 = Function0(Var7, Var8)
```

(b) Example code

Figure 8: Experiment 5 Results

Experiment	1	2	3	4
1	N	N	N	N
2	Y	N	N	N
3	Y	Y	N	N
4	Y	N	N	N
5	Y	Y	N	N
6	Y	N	N	N
7	Y	N	N	N
8	Y	Y	Y	Y
9	Y	N	N	N
10	Y	Y	N	N
Total	9	4	1	1

(a) Table of results



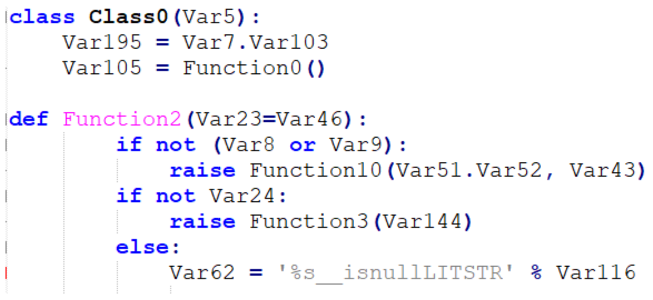
```
def Function1(Var108=Var8, *Var76):
        pass
    Var62 = Function3()
    Var446 = lambda Var11='fieldLITSTR', *Var76: Var2
    Var22 = Var2
    Var392['path_suffixLITSTR'] = Var19

    def Function2(Var4, Var75, Var23, Var8, Var54=100):
        return Var77
```

(b) Example code

Figure 9: Experiment 6 Results

Experiment	1	2	3	4
1	Y	Y	N	N
2	Y	N	N	N
3	Y	Y	N	N
4	Y	Y	N	N
5	Y	Y	Y	Y
6	Y	N	N	N
7	Y	Y	N	N
8	Y	N	N	N
9	Y	N	N	N
10	Y	Y	N	N
Total	10	6	1	1

(a) Table of results



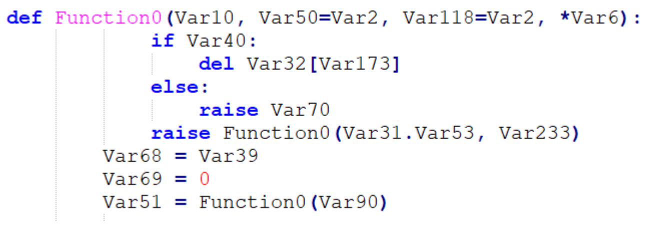
```
class Class0(Var5):
    Var195 = Var7.Var103
    Var105 = Function0()

def Function2(Var23=Var46):
        if not (Var8 or Var9):
            raise Function10(Var51.Var52, Var43)
        if not Var24:
            raise Function3(Var144)
        else:
            Var62 = '%s__isnullLITSTR' % Var116
```

(b) Example code

Figure 10: Experiment 7 Results

Experiment	1	2	3	4
1	N	N	N	N
2	Y	Y	N	N
3	Y	N	N	N
4	Y	N	N	N
5	Y	N	N	N
6	Y	Y	N	N
7	Y	Y	N	N
8	Y	N	N	N
9	N	N	N	N
10	Y	N	N	N
Total	8	3	0	0

(a) Table of results



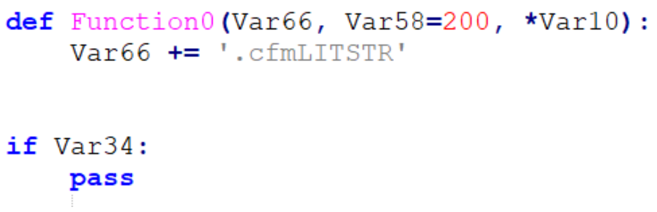
```
def Function0(Var10, Var50=Var2, Var118=Var2, *Var6):
            if Var40:
                del Var32[Var173]
            else:
                raise Var70
            raise Function0(Var31.Var53, Var233)
        Var68 = Var39
        Var69 = 0
        Var51 = Function0(Var90)
```

(b) Example code

Figure 11: Experiment 8 Results

Experiment	1	2	3	4
1	Y	Y	N	N
2	Y	N	N	N
3	Y	Y	N	N
4	Y	Y	N	N
5	Y	N	N	N
6	Y	N	N	N
7	Y	Y	N	N
8	N	N	N	N
9	Y	N	N	N
10	Y	Y	N	N
Total	9	5	0	0

(a) Table of results



```
def Function0(Var66, Var58=200, *Var10):
    Var66 += '.cfmLITSTR'


if Var34:
    pass
```

(b) Example code

Figure 12: Experiment 9 Results

5.3.2 Experiments 4-6

Experiments 4-6, presented in Figures 7-9, show an interesting set of results. At this point the change in corpus size appears to have less of an effect on the model results. As the variable maximum has increased, the overall effectiveness of the model has been reduced, except for in the case of the large corpus.

5.3.3 Experiments 7-9

With experiments 7-9, presented in Figures 10-12, only one generation was able to produce an output, the same generation was able to produce an output that depended on the input. The variable maximum at this stage appeared to have removed the effects of increasing the corpus size, in terms of the measurements we were taking from the code. Many of the code generations in this stage were very long and had multiple class and function declarations.

6 Results and Model Evaluation

6.1 Effect of the Corpus Size

The introduction of more code for the model to train on increases the complexity of the programs generated, as the code is being spliced together by the model is coming from an increasing number of sources. There is less cohesion in the model and subsequently the inputs are not being processed. The code formed appears to be many small statements each doing their own tasks; they are not forming together to create an overall program. At very low corpus sizes however the code generated was too similar to the code found in the corpus. Once the initial node had been generated the model was much more likely to follow the probability chain that leads to a program similar to that of one in the corpus. The model is also much more sensitive to the data in the corpus when the corpus is smaller. Since it is more likely for the code from any given file to be generated, any programs in the corpus with poor data for generation will have a much larger effect on the end result. The main effect of increasing the corpus size was an increase in the number of runtime errors occurring in the program. This can be seen in the results presented in Figure 13. With more code to choose from, the chance of a code block to be in the right context to run is reduced, as it is unlikely that any of the generated surrounding code is doing anything similar to the code block. Surprisingly, the corpus size had little effect on the size of the programs being produced. It was equally possible for a program to be multiple functions or just a few lines of code no matter the corpus size.

6.2 Effect of the Variable Maximum

Altering the maximum number of variables in the generated program has the biggest effect on the models capability to prove all of the research hypotheses. Whenever the number of variables in the program is increased, no matter the corpus size, the chance of the model being able to produce an output that changes depending on the input is reduced. This can be seen in the results presented in Figure 14. This result follows the expectations from the model. As the number of variables increases, the chance of a variable being used more than once is reduced. This means that the result of any statement at the start of the generated code that is stored in a variable is unlikely to be used again. Even in the smallest corpus size where the total number of variables is not particularly large there is a dramatic fall off in the models capability. The obvious answer here may be to always limit the number of variables that the model will generate, as this will produce programs where the variable changes can persist through statements. This however can pose an issue with limiting program complexity. More complex programs require more variables; by limiting the number of variables the programs generated may work better but the program complexity is limited. What is ultimately required from the model is an unlimited

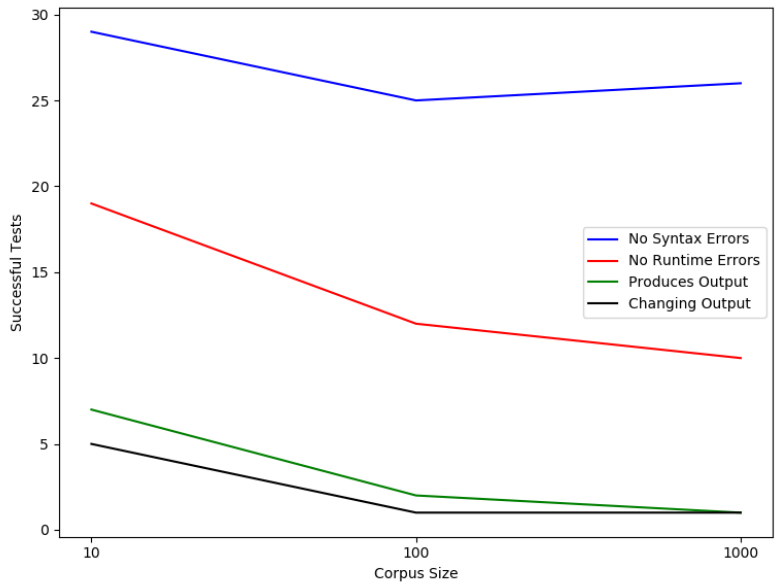


Figure 13: Effect of Corpus Size on Successful Tests

number of variables, to allow for maximum complexity, whilst assuring that the variables will be used cross statement so that the information they hold is not useless or lost.

Another effect of changing the maximum number of variables was a change in the size of the programs being generated. At smaller values the programs were much longer. The model would have more candidates to choose from after each variable, this is because the variable renaming would rename many variables in the corpus to be the same variable in the model. Because of this there is less chance of a variable leading to the end of the generation, so the programs are often longer.

6.3 Results Evaluation

6.3.1 Achieving the Research Hypothesis

From the results it is clear that the first research hypothesis has been achieved. The model is able to very consistently generate code that is syntactically correct. This can be seen in the high values presented in Figures 13 and 14. Only in certain edge cases does the model fail to generate syntactically correct code. The model design limits the node generations probabilities to depend only on the parent node. When generating a node, the model is not concerned with the parents parent. In the case, for example, of a return statement generated within an if statement, the if statement may not be within a function. The return statement is therefore not in a function resulting in a syntax error. This is not prevented by the model since it is only concerned with the return statements parent, and since the return will have been seen in the if statement within the training data it is a perfectly valid generation. Only after extensive testing with a variety of corpus sizes and variable maximum was this issue found. This further highlights that the context of generation is as of as much importance as the n-gram probabilities.

Secondly, the model has been shown to generate code that has no runtime errors. In many cases the code generated was able to run without issue. When runtime errors did occur, the main cause was calls to functions or methods that didnt exist. The larger the training corpus the greater the number

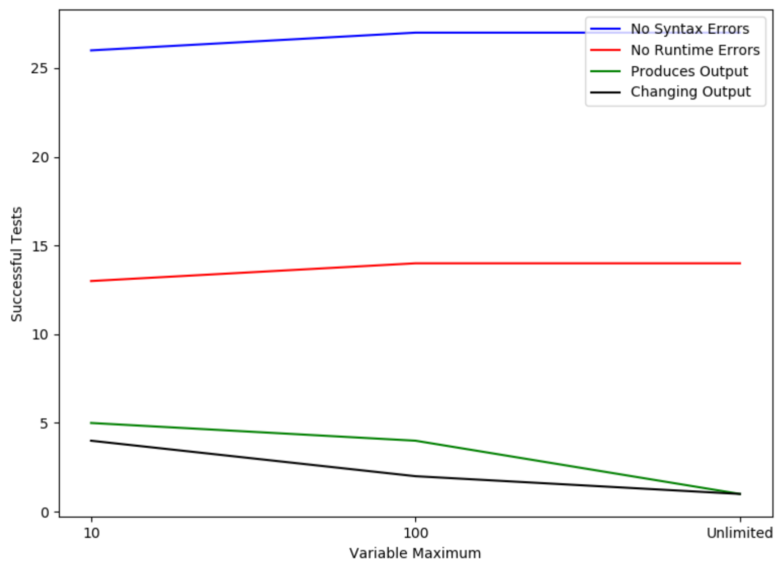


Figure 14: Effect of Variable Maximum on Successful Tests

of runtime errors, as seen in Figure 13. This was caused by an increase in the code complexity, which was caused by an increase in corpus size. The model and learning was not at fault here. The main issue came with the probabilities within the model not being altered based on the current context.

Finally, hypotheses (3) and (4) were very rarely proved by the model; only in a very few cases did the model successfully produce an output or produce an output that changed with the input. The cases of success were more by chance of the n-gram model generating the correct code, rather than the model actively using its logic to produce the program outputs. Whilst hypothesis (3) and (4) have been achieved in a few cases, they have not been achieved enough to be proven. The believed reasoning as to why the model failed to prove these hypotheses is discussed in the model evaluation section.

Attempts at program generation in literature have attemped to measure the success of the model by omitting nodes and having the model attempt to generate them. These tests are in essence a test of the model's ability to generate syntactically correct code without runtime errors. Previous values of 50% have been achieved at this [16]. From the results, the average success rate of correct syntax is 88.89% and the succes rate of no runtime errors is 45%. These figures are similar to that of those found in literature and show the importance of the context function, since the correct syntax percent is much higher.

From the experimentation, it has been found that the model is most effective when the corpus size and variable maximum values are kept low. The model has shown the best results at achieving the research hypotheses at the values of 10 and 10 for the corpus size and variable maximum value. Although these values allow for the most success in proving the hypotheses, the programs generated will be limited in complexity and function.

6.4 Model Evaluation

6.4.1 The Base N-gram Model

The n-gram model implemented proved to be very successful in generating the program code to achieve the first two hypotheses. The implementation of the n-gram model over the JSON AST proved successful; it can correctly capture the syntax of the code and generate varied sections of code, formed together from the code in the programs of the training corpus. The generated code appeared human-like, meaning that it was written in a style where it would have been feasible for a human programmer to have written the code. This is very important when relating back to the project motivation discussed in the introduction, and this is one of the main benefits of generating code in this way. The renaming of variables was highly effective in causing the programs to be spliced together, resulting in complex and varied sections of code that were unlike sections seen in the original corpus. The n-gram model did however struggle to handle the depth of the AST, meaning that the program generation had to be halted and restarted sometimes. This was particularly a problem at greater corpus sizes. Since there were so many candidate options for the model to choose from the probability of there being no more nodes became very small.

6.4.2 The Importance of the Context Function

The additions made to the code to add context to the generations proved successful. Limiting the sibling types removed many type errors; the model was unable to generate invalid operations between variables. The renaming of functions and function calls based on what had been declared made it possible for functions calls to be successful. Without the renaming, the probability of a successful name match was too slim. Whilst some additions were made to the model in the form of a context function, for the model to be highly successful in achieving all four hypotheses many more additions have to be made. Many of the issues in the model generations come from the model not knowing the context of the surrounding code. When writing code, programmers will know the requirements for a statement they are writing to work. For example, if they are writing a function call it must be declared, or any variables being used must be declared and variables being used must have correct type. This knowledge of what is required around the statement is what causes many of the runtime problems in the program code.

Often, variables would be assigned values globally that would not match the value of the variable required in the code, for example, a statement would try to access an attribute a on variable that does not have attributes. Even though functions and function calls were renamed, some programs would contain no function declarations. Any function calls in the code would therefore cause runtime errors. This also caused the problem of functions being declared but not being called. As there was nothing to direct the model to generate code outside function or class declarations, except from code seen in the corpus, the space at the program level was often empty. This problem is exacerbated by the standard program design that would have been present in the corpus. Traditionally, program code in larger projects is written across multiple files. Many of the files will simply contain function declarations that are used by a main file. The probability of selecting files from the corpus that contain code inside the main body is therefore relatively low, leading to generated code that is more likely to have no code in the main body. Since code outside the main body is limited, any programs with code in the main body would often be short and would rarely contain function declarations.

6.4.3 Generating Outputs and Controlling the Code

Whilst for a program to be useful it must generate some form of output, generating code that is able to do this is very challenging. Although aiming to have the generated code perform a specific task is out scope for this project, guaranteeing an output that depends on the input, or guaranteeing any type of output at all, is a very basic level of controlling the code that is being generated. Consistently generating code where the inputs are used for more than a single statement, or beyond that where the inputs have an impact on the results, is very challenging. When the maximum number of different variables in the code is reduced this is much more likely, simply because the chance of a variable being generated again is much higher. As the corpus size increased the ability to generate inputs based on the outputs decreased. Even at lower maximum variable amounts, the model was unable to generate code statements that were trying to do the same thing. This was much more likely at lower corpus amounts as code being generated is more likely to originate from the same file. As discussed previously the idea of what an output is was changed for the purpose of the experiments. The number of possible candidates that actually produce an output, such as a print node, is very low in the corpus, the probability of one of these nodes being generated is therefore simply too low. The difficulty of controlling the generated code, alongside the issues brought about by the context function, resulted in the model being unable to prove hypotheses 3 and 4.

7 Critical Assessment of the Project

7.1 The Approach to the Problem

The initial problem of choosing a probabilistic model and implementing the chosen model was approached well. The current literature on the topic was reviewed in depth and a model was chosen that was capable of proving the research hypotheses. The model was implemented through incremental changes that were as a result of the development testing. Through these incremental changes, the main n-gram model was developed to a state where it was able to fulfil its requirements in the project.

However, the importance of the context function to be implemented alongside the model was underestimated. This is what has led to the model being unable to consistently prove research hypothesis (3) and (4). Whilst the beginnings of the context function are implemented in the model, and it has been implemented according to the specification and the design, additions would be required beyond the specification for the model to be highly successful in proving all research hypotheses. The model at its current state is at an excellent base point for improvements and adjustments to include more context in the program generation.

7.2 The Overall Process

The development plan of implementing the basic prototype model, testing this with simple data and then slowly improving the model whilst increasing the test data complexity allowed for incremental improvements, where the changes made to the model were in response to the ongoing development testing. It was particularly important that the tests were kept simple during development as using large amounts of data would have made testing the model very difficult.

The experiment design allowed for the effects of the model, corpus size and variable maximum to be tested independently. Since there is more than one independent variable per test, it is important to run every corpus size against every variable maximum, so that the effects of each can be differentiated. To further investigate the effects of these variables on the model, more values could have been used at smaller increments to find a more accurate result of where the model is most effective.

8 Summary and Conclusion

This project has shown the development process of designing and implementing a probabilistic model to generate python program code. Firstly, the n-gram model was decided as the method of choice to be used for the project. Its ability to capture dependencies within the code at a longer range than a PCFG model, whilst being simpler and requiring significantly less time to train than a neural network made it the most suitable for the problem. ASTs were used alongside the n-gram model as a way to represent the python code as they are efficient and useable. For the project four research hypotheses were defined. (1) Does the program run without syntax errors? (2) Does the program run without runtime errors? (3) Does the program produce an output? (4) Does the program produce an output that depends on the input? These would be used to evaluate the success of the model during experimentation. The design of the project was based on implementing the n-gram model across a tree structure. The bigram was formed by the parent, last item and position of the node in the tree. Implementing the n-gram model in this way proved to be very successful; the model was able to capture the syntax of python and generate the code.

This design was first implemented as a prototype. The initial functionality of the model allowed for training, code generation and execution to test the programs implemented. This prototype was then extended to further improve the quality of the code. Firstly, variable renaming proved effective in increasing the interactions between statements drawn from different programs in the corpus. Altering function declarations and calls allowed the program to interact with code that it had generated. The context function implemented restricted variables used depending on their types; this fixed the issue of generating invalid operations between variables.

Upon completion of development, experimentation showed that the model proved hypotheses (1) and (2) with (3) and (4) being proved in some cases. Testing followed the method of generating code across three corpus sizes (10, 100, 1000) and three variable maximum values (10, 100, unlimited). The variable maximum values indicated the number of renamed variables, which would be in the model after training. Since the model was only able to prove (3) and (4) with very low corpus sizes and variable maximum values, it was taken that they had not been proven. The model was shown to be most effective when the corpus size and variable maximum values were at their lowest. The model showed much success in generating syntactically correct code, that is able to run without runtime errors across many corpus sizes and maximum variable counts. Across all tests the model generated syntactically correct code in 88% of generations and code without runtime errors in 45%. These results proved the success of the n-gram model in generating python code. Overall, the project has been successful in showing the capability of the n-gram model to generate program code. It has been very succesful in generating syntactically correct code, which runs without runtime errors. With additions to the context function and logic surrounding the model it will be able to generate more advanced code and be able to fully prove all the research hyptotheses put forward by this project.

8.0.1 Future Work

The main extension that can be made to the current model is the introduction of more context to the code generation. As discussed previously, this will enable the model to generate code that has fewer runtime errors and can aid in directing the code to use any functions that it declares. Further on from this, the model could be altered so that it attempts to generate programs to solve a specific problem. This could be either a pre-set problem that is hard coded into the model, or a problem input by the user. This is a very advanced addition to the current model and leads to some of the real-world implementations discussed in the introduction. Being able to use the model to generate specific programs will enable it to be used in development and IDE tools to speed up the development process.

References

[1] Astor 0.7.1. https://github.com/berkerpeksag/astor, 2019.

[2] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[4] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942, 2016.

[5] Geoffrey C Fox and JG Koller. Code generation by a generalized neural network: general principles and elementary examples. *Journal of Parallel and Distributed Computing*, 6(2):388–410, 1989.

[6] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.

[7] Frederick Jelinek, John D Lafferty, and Robert L Mercer. Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding*, pages 345–360. Springer, 1992.

[8] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, pages 1–10, 2003.

[9] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*, 2015.

[10] Irene Langkilde and Kevin Knight. The practical value of n-grams is in generation. *Natural Language Generation*, 1998.

[11] Christina Lioma and CJ Keith van Rijsbergen. Part of speech n-grams and information retrieval. *Revue française de linguistique appliquée*, 13(1):9–22, 2008.

[12] Kenneth C Louden. Compiler construction. *Cengage Learning*, 1997.

[13] Chris Maddison and Daniel Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning*, pages 649–657, 2014.

[14] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[15] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

[16] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *ACM SIGPLAN Notices*, volume 51, pages 731–747. ACM, 2016.

[17] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *ACM SIGPLAN Notices*, volume 51, pages 761–774. ACM, 2016.

[18] Raúl Rojas. *Neural networks: a systematic introduction.* Springer Science & Business Media, 2013.

[19] Shigeyoshi Tsutsui. Probabilistic model-building genetic algorithms in permutation representation domain using edge histogram. In *International Conference on Parallel Problem Solving from Nature*, pages 224–233. Springer, 2002.