



University
of Glasgow | School of
Computing Science

Honours Individual Project Dissertation

DEEP RECURRENT NETWORKS FOR LANGUAGE CONDITIONED CODE GENERATION

Carlos Gemmell
March 7, 2019

Abstract

Tools capable of code generation have the capacity to augment programmers capabilities drastically. However, the field of software engineering provides primitive tools with limited search and completion functionality. In this paper we compare current methods from both code retrieval and machine-learned code generation models for the task of code generation, and propose a new approach combining these two techniques. We use existing standard benchmark collections, including Django and CoNaLa. In addition, we develop new human generated datasets with a focus on short, realistic snippets of code. We also study methods for generating large-scale synthetic code collections needed to train neural translation models. We experiment with domain transfer methods across the various code collections. We find that current retrieval based algorithms are significantly more effective than neural translation-based approaches. We also analyze our findings and demonstrate the limitations of current models, suggesting new areas for future improvements.

Acknowledgements

I would like to thank my supervisor Jeff Dalton for his support in this project. His input was invaluable and I enjoyed our work throughout the year.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Aim	1
1.3	Outline	2
2	Background	3
2.1	Sequential models	3
2.1.1	RNNs	3
2.1.2	Gated RNNs	4
2.1.3	BiRNN architectures	4
2.1.4	Mult-RNNs	4
2.1.5	Encoder-Decoder architectures	5
2.1.6	Attention	5
2.2	Code generation	6
2.2.1	Hybrid Models	6
2.3	Retrieval	6
2.3.1	TF-IDF	7
2.3.2	Cosine Similarity	7
2.4	Related work: Sourcerer	7
2.5	Krippendorff's agreement measure	7
2.6	BLEU	8
3	Analysis/Requirements	9
3.1	Overview	9
3.2	User Personas	9
3.2.1	Jack	10
3.2.2	Beth	10
3.2.3	Andrew	10
3.3	Functional Requirements	10
3.3.1	Must have	11
3.3.2	Should have	11
3.3.3	Could have	11
3.3.4	Won't have	11
4	Design	12
4.1	Datasources	12
4.2	Algorithms	12
4.2.1	Retrieval	12
4.2.2	Generative	13
4.2.3	Hybrid	13
4.3	User Interface	13
4.4	Model comparison	13
5	Implementation	14
5.1	Data	14

5.1.1	Data Storage	14
5.1.2	Description tokenization	14
5.2	Vocabulary and code tokenization	15
5.3	Synthetic data generation and augmentation	17
5.4	Technologies used	17
5.4.1	Tensorflow	17
5.4.2	Tensorboard	18
5.4.3	Scikit-learn	18
5.4.4	Numpy	18
5.4.5	Google Cloud Compute	19
5.5	Algorithms	19
5.5.1	Retrieval	19
5.5.2	Generative	20
5.5.3	Hybrid: SeekNMT	23
5.6	Metrics	24
5.6.1	BLEU	24
5.6.2	User Study	25
5.7	User and Program Interfaces	26
5.7.1	Server	26
5.7.2	User Interfaces	27
6	Evaluation	28
6.1	Datasets	28
6.1.1	Django	28
6.1.2	Docstring	29
6.1.3	CoNaLa	29
6.1.4	Synthetic	30
6.1.5	Dataset comparisons	31
6.2	Testing	31
6.2.1	Dummy prototype dataset	31
6.2.2	ConCode	31
6.3	Training	32
6.3.1	Retrieval	32
6.3.2	NMT and SeekNMT	33
6.4	Metrics	34
6.4.1	In-domain: Dev BLEU	34
6.4.2	Out-domain: Stack BLEU	34
6.4.3	Manual Relevance Assessment: Precision@1	35
6.5	Results	36
6.5.1	Baseline Retrieval and NMT results	36
6.5.2	Manual assessment	38
7	Conclusion	39
7.1	Summary	39
7.2	Contributions	39
7.3	Reflection	39
7.4	Future Work	40
	Appendices	41
A	Appendices	41
	Bibliography	48

1 | Introduction

1.1 Motivation

Programming is a task that doesn't come naturally to us. We don't think in code yet we have restricted machines that understand a very specific language. This language tells the machine what to do. It is our job as programmers to bridge the gap between our thoughts and the program we wish to run on the machine.

Programming languages demand concentrated attention and outstanding knowledge of syntax and grammar of the underlying libraries and algorithms. Since the proliferation of the Internet, search engines like Google and programming forums like StackOverflow have allowed programmers to share knowledge for common programming tasks by sharing solutions to common problems in code. These usually take the form of a question to which other more experienced programmers write a solution in code, then comment, and discuss for further clarifications. Programmers require parallel knowledge of many different programming paradigms, libraries and techniques to effectively write code. The sheer amount of structured information required is often too much to keep in one's head and means online searches to these forums for library examples or clarifications on syntax are frequent. This process lengthens the time from idea to implementation and can also make the programmer lose focus as they veer off on tangents to search for forgotten syntax. Code generation techniques tackle this issue by allowing the programmer to write their implementation as a sentence in natural language, which is then interpreted and converted by an algorithm into the code they described. This would reduce cognitive load by avoiding the need to remember exact syntax. Avoiding frequent searches has the possibility of transforming the programming landscape to new levels of efficiency and creativity.

1.2 Aim

The field of code generation is developing fast with recent research frequently pushing the boundaries. The aim of this project is to explore and evaluate current techniques in code generation from natural language descriptions. Code generation is currently being approached in ways similar to other language or retrieval tasks with their evaluation methods following suit. We set out to gain a deeper understanding of how these models attempt this challenging task by building and training several systems and seeing the effect of data, and the underlying algorithm on the quality of generated code. We evaluate these models with metrics standard to research in the field, and conduct our own user studies on programmers to assess the effectiveness from a more realistic point of view. We package our code generating algorithms into an application capable of receiving code descriptions from an Integrated Development Environment (IDE) returning back the generated code.

We define the task of Code Generation from Natural Language as follows: Given a topic description, T , the goal is to generate the single most relevant snippet, S , of code that satisfies the topic description.

To perform this task we commonly formulate it as follows: **Input:** NL tokens from T are split into a sequence t_i , i denoting the position of the token in the sequence.

Output: Code tokens split into a sequence s_i

We note that the result can come from either retrieval (existing code) or be generated by a generative model, such as the translation models we experiment with. The outputs are short snippets – equivalent to a small line (or lines) of code. This is roughly analogous to generating or retrieving code at the level of sentence or paragraph.

We hope to answer the following questions with this research:

- Which algorithms most effectively generate useful code?
- Can a combination of multiple approaches yield improvements in code generation over current techniques?
- Are current evaluation methods suitable to characterize model performance?

1.3 Outline

In this work, we take steps to investigate current methods in code generation and evaluate their performance using several metrics. We structure the rest of the document as follows:

Chapter 2: **Background**

This chapter discusses current related systems and the background knowledge needed.

Chapter 3: **Analysis and Requirements**

This chapter discusses the project specification and how the requirements were gathered.

Chapter 4: **Design**

This chapter discusses design of the solution, presenting an overview of the project structure, and covering high level design decisions.

Chapter 5: **Implementation**

This chapter discusses the implementation

Chapter 6: **Evaluation**

This chapter discusses the evaluation of the final system over several benchmarks.

Chapter 7: **Conclusion**

This chapter provides a summary of the proposed system and discusses our findings.

2 | Background

In this section we cover research and related problems relevant to code generation from natural language descriptions.

2.1 Sequential models

2.1.1 RNNs

The field of code generation has been heavily shaped in recent times with the advent of new machine learning techniques such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks. RNNs are a way of processing sequential information of variable length. Originally developed by Rumelhart David E. et al. (1986), RNNs transformed the uses of neural networks to handle variable length input.

Traditional neural network architectures feature a fixed input format. They operate on the premise that the input data will keep a similar shape and it is the networks task to find patterns in the data by tweaking its internal variables through backpropagation (Werbos (1990)) to minimize the final cost function. Neural Networks (NNs) are well suited to images for instance where all the information in the scene is captured in a single fixed size frame. Recurrent Neural Networks differ from traditional feed forward networks by feeding the output of the network to the next input. This conditions the next output of the network to make decisions based on the previous output thus keeping information from past observations.

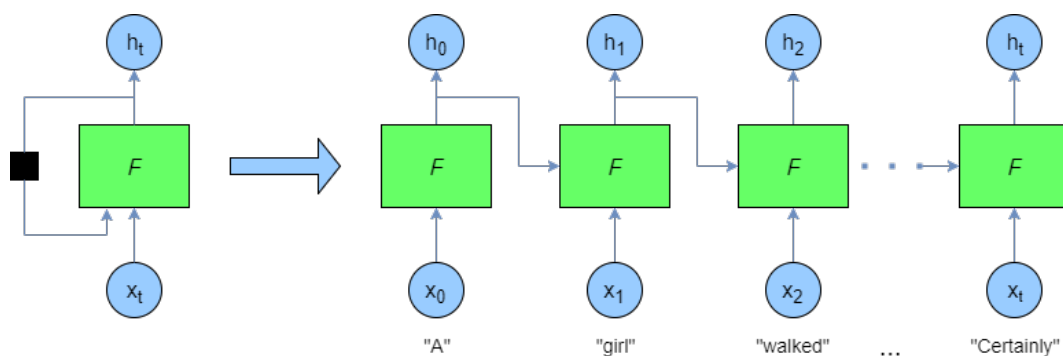


Figure 2.1: Recurrent Neural Network processing a text sequence of variable length noa (2018)

In practice RNNs suffer from difficulty learning long term dependencies, something crucial in sequential data such as text (Hochreiter (1998); Hochreiter and Schmidhuber (1997)). This difficulty arises from a decaying gradient when propagating the error of an output back in the sequence. This is most notable with long term dependencies and means that converging to a global minimum on the cost function is very difficult.

Most modern sequential models have moved away from the original RNN in favor of more stable variants such as LSTMs (Hochreiter and Schmidhuber (1997)) and GRUs (Gated Recurrent Units) by Cho et al. (2014a). These differ most notably by being able to selectively add to the memory state from the input. This is different from RNNs since the state they pass on is a forced new combination of the input and the previous state.

2.1.2 Gated RNNs

LSTMs in particular hold time dependant elements in a separate hidden state that can be modified through dedicated "remember" or "forget" gates. These gates take the input state and are element-wise scalar values that apply only to the memory. This memory is then passed from one time step to the next with no other transformation. This leads to long term dependencies being unaffected by irrelevant input and makes the propagation of errors simpler.

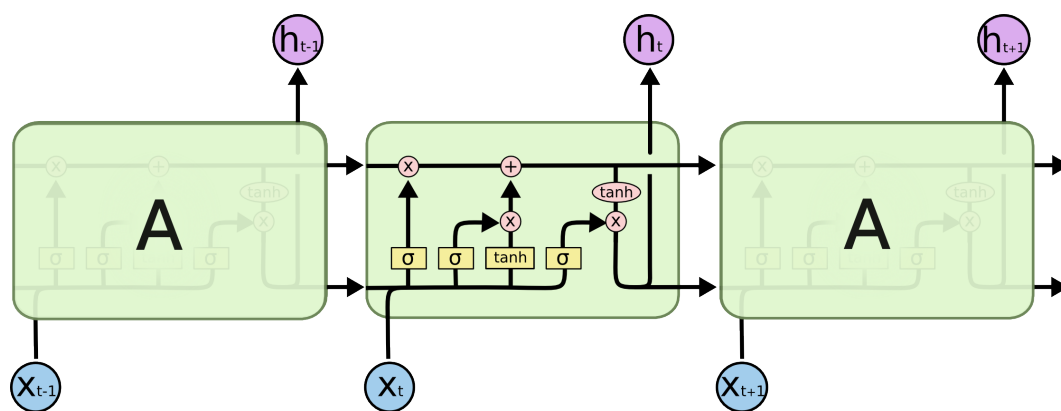


Figure 2.2: Example of an LSTM cell processing a sequence of variable length from Olah (2015)

2.1.3 BiRNN architectures

Bidirectional RNNs (BiRNNs) are variation on the Multi-RNN cell (Schuster and Paliwal (1997)). These consist in sending the input sequence to two RNN cells simultaneously, however the time component for one of them is reversed, effectively passing the input sequence backwards. This shows improvements in encoding meaning into the memory state since both cells receive identical input.

2.1.4 Multit-RNNs

LSTMs, RNNs and GRUs can be considered single modules inside a larger network. Much like how deep neural networks stack simpler layers such as convolutions, or fully connected layers, Recurrent Neural Network architectures can benefit from similar deeper architectures. Liu et al. (2016) show how LSTMs can be stacked into a larger cell. Here the output from the cells that take in the feature data is passed as the input to another cell for the same time step. This can be repeated for n LSTM cells making an n layer Multi-LSTM stack. These show greater generalization over simpler stacks at the cost of model complexity in numbers of parameters to train.

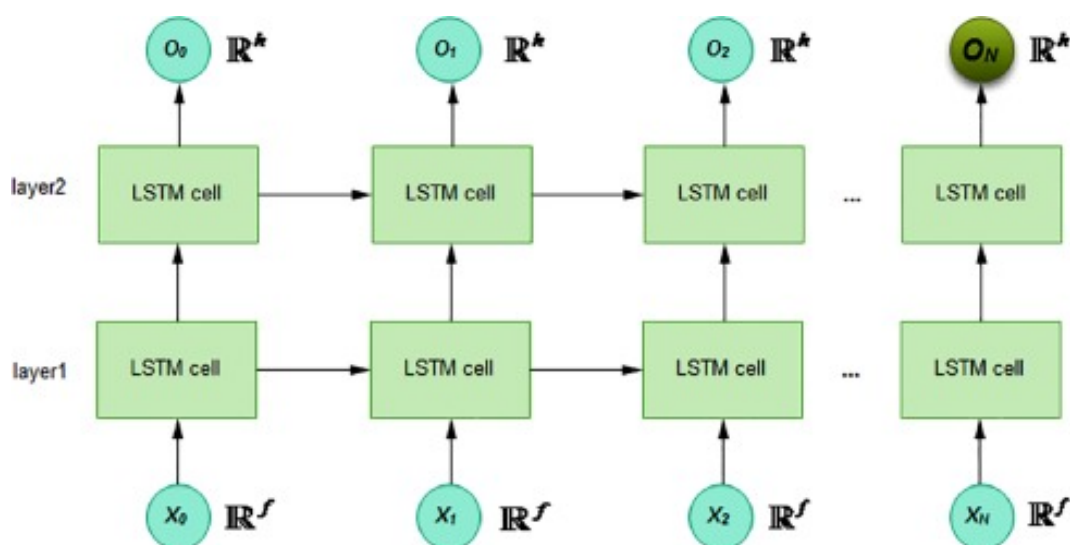


Figure 2.3: Multi-LSTM cell processing a sequence of variable length Zhao et al. (2018)

2.1.5 Encoder-Decoder architectures

RNNs and its variants serve the purpose of encoding sequential information, often of variable length into a fixed length context vector: the last hidden memory state. This can be thought of as the vector representation of the meaning of the sequential data. This fixed length state can then be passed as input to a separate module according to the necessary output for the mode. ? show how text classification can be performed, where the context vector is passed to a fully connected classifier network to label text based on its context.

Encoder-decoder networks take the hidden state from the RNN and pass it as the initial state of a second RNN module. This second one is called the decoder. At each time step in the decoder, the produced hidden state is used as input to a classifier network to select from a pool of output tokens. Sutskever et al. (2014) use this architecture to make a model capable of English-German translation.

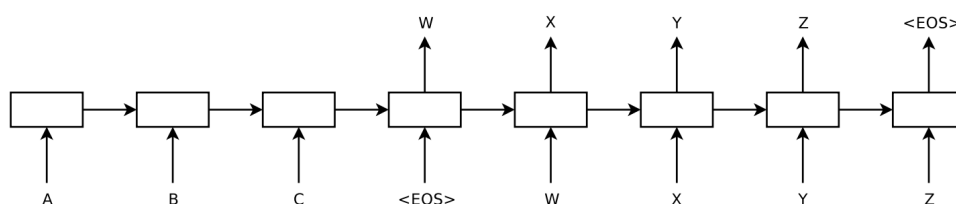


Figure 2.4: Sequence being transformed using an Encoder-Decoder network Brownlee (2017)

2.1.6 Attention

The above sequence to sequence models (Seq2Seq) take only the final state from the encoder to start the decoder. This means that all the information that could possibly be relevant to the

output sequence needs to be encoded in that one state, and that long term dependencies survive the encoding process and have been preserved. Bahdanau et al. (2014) improve models in this regard by aligning the input sequence with the output.

Attention is a process that occurs during decoding. The input states are passed to an additional layer that produces a weighting for the input states when producing a new token. This effectively acts as a shortcut for a relevant initial state to influence an output state that is far away. This is opposed to the standard backpropagation of error in an encoder-decoder that would need to propagate all the way through the sequence.

When tokens in the input are relevant to the decoding of a new input the intention is that the input hidden states from those relevant tokens are weighted highly so that they influence the decoder heavily. This has shown to improve recall of long term dependencies.

2.2 Code generation

Most recent work treats Code Generation as a Machine Translation task and applies translation models, such as encoder-decoder networks. First attempts involved input and output as linear sequences Sutskever et al. (2014). Words are tokenized and passed to an embedding layer, and are then encoded into a fixed size latent space with one or more memory modules. Tokens in the target language are then generated from this latent space. This allows for variable length sequences to be taken in and generated.

Following this, others suggest a way of generating code to take advantage of the programming language's inherent structure, representing the output as a series of construction decisions. Rabinovich et al. (2017) take it a step further by using Abstract Syntax Trees (AST) in the decoding process. Their network generates a tree structure in a top-down manner by adding atomic operations which build the tree, keeping information from parent and sibling nodes. This technique is well suited for the task since programming languages follow a tree structure with intermediate nodes holding functions, and leaf nodes holding primitives like Strings and Ints.

2.2.1 Hybrid Models

However, the work mentioned above does not include any context such as already existing code or variables when feeding the source. When writing code that integrates with a larger program, it usually depends on variables or methods in said source. Iyer et al. (2018) attempt to solve this by appending already existing code into the input vector, and changing distinct variable names to universal positional tokens combined with AST decoders.

Their method for adding information to the input sequence involves adding a series of separator tokens specific to the section they are separating. These special tokens serve to give the attention mechanism queues to know when each section starts.

2.3 Retrieval

Retrieval models are well established in the field of Code Improvement. Many attempts emphasize helping programmers debug programs, and remove duplicate or similar code by identifying close matches from different sections in source code. Early approaches Jeng and Cheng (1993) rely on highly structured formal methods to convert queries such as "find all functions that contain a variable arr" into a specialized query language to search for exact matches. Mishne and Rijke

(2004) propose code snippet retrieval by forming unstructured queries over code and use a "fuzzy" approach to help programmers find similar snippets to their query. These approaches attempt to search over the target code to find relevant results. Sindhgatta (2006) employs a different approach by querying over code authors' annotations to retrieve relevant snippets.

2.3.1 TF-IDF

Term Frequency over Inverse Document Frequency (TF-IDF) is a method for representing a sequence of tokens as a vector where each dimension of said vector is a unique term in the corpus of all documents. As described by Hiemstra (2000), TF-IDF is a specific weighting of vocabulary based on the presence in the document and penalised by the frequency the terms appear in the corpus. Words that appear often in every document are given less importance and contribute less to the "aboutness" of a document.

2.3.2 Cosine Similarity

Cosine similarity is a measure indicating similarity between two non zero vectors with the cosine angle being minimum when the vectors are aligned.

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|} \quad (2.1)$$

Figure 2.5: Mathematical expression depicting cosine similarity where x and y are non zero vectors

2.4 Related work: Sourcerer

Sourcerer is a text editor plugin for Atom which allows you to select text you have written in the editor and retrieve the code from the best matching answer on StackOverflow. The editor interface is intuitive and has you select the text you want to use for the query and then substitutes it with the associated code. It enables the user to write the description without needing to click on unnecessary buttons lengthening the process from idea to result. Results seen in Figure 2.6, are returned by querying the StackOverflow web API's and sorted based on relevance allowing you to pick the best match for your needs.

2.5 Krippendorff's agreement measure

Krippendorff's agreement measure, Krippendorff (2011), is a metric (ranging from 0 to 1) indicating agreement when collecting samples from multiple participants. It is important to measure agreement for method validity. Low measures (<0.7) indicate participants are unable to come to similar conclusions when evaluating the same question meaning claims based on the results have little basis. Higher values (>0.7) are more worthy of note as users come to similar conclusions.

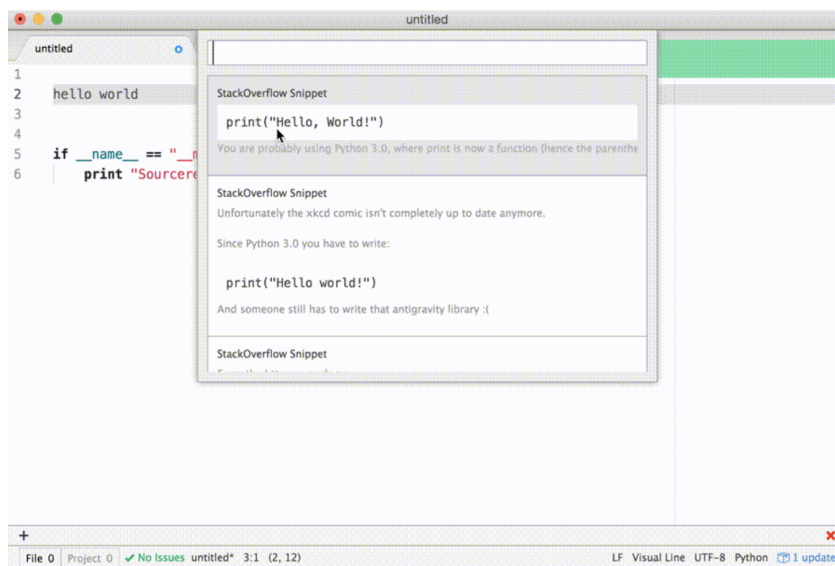


Figure 2.6: Example code sourcing from a description using Sourcerer

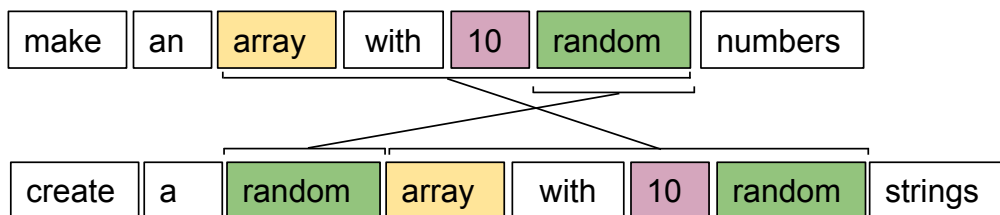


Figure 2.7: Visualization of BLEU n -gram locations between two sentences

2.6 BLEU

Made by Papineni et al. (2002), BLEU is an automatic metric similar to Unigram Precision, however it takes into account the order the words appear in more than the previous metric by checking the presence of n -grams in either sentence. The location of n -grams does not matter in the sequence as long as their presence is noted. BLEU can be measured by having multiple reference sentences and comparing n -gram's presence from the predicted sequence in any of the references.

BLEU has seen a surge in use since its release, notably in the field of machine translation, Wu et al. (2016). And is currently being used as a standard metric for performance in the field of code generation Iyer et al. (2018); Rabinovich et al. (2017). Although this automatic metric is convenient, it can sometimes not give a full picture for the real quality of translation Callison-Burch et al. (2006)

Bleu's inability to distinguish between randomly generated variations in translation hints that it may not correlate with human judgments of translation quality in some cases.

3 | Analysis/Requirements

In this chapter we cover the project requirements and how we arrived at them.

3.1 Overview

We set out to improve the experience of programming by lowering the cognitive load of simpler low level tasks in Python.

We choose Python for the following reasons:

1. Syntax is simple and readable
2. Statements can be written concisely such that a single simple line can have a meaningful impact on a program
3. The environments where Python is used in real world systems are often not as critical and sensitive to errors as other languages such as C.
4. Many datasets involving Python code are available publicly from previous research.

System safety is an important consideration since we are using algorithms whose decisions are conditioned on the data that is presented to them. This can lead to unexpected results that are difficult to identify. We hope to minimize the risk of our system being the cause of critical error by having our target user base be research oriented or involved in low risk activities such as hackathons.

Programming requires parallel knowledge of multiple libraries and syntax rules that are not used often enough to be committed to long-term memory this means having to search frequently for the answer on sites such as StackOverflow, Quora, Reddit, StackExchange...

We aim to help programmers with:

- Single line or very simple multi-line statements
- Standard library functions
- List comprehensions
- String Operations
- Simple program control flow

3.2 User Personas

These are examples of the kind of people that would use the system:

3.2.1 Jack

Jack is a computer science student who is taking part in a hackathon. One of the given challenges is making a game in python. Jack wants to start the game by creating an algorithm to simulate the collisions and gravity between objects in the game. However, his courses don't usually cover game physics so he struggles to write the code. He writes pseudo-code for his algorithm specifying vaguely what inputs there are, where for-loops and comparisons are, what general operations are made and which variables are returned. He feeds the pseudo-code as a string to the Desc2Code algorithm and receives working or closely working code that reflects his original design in an understandable way. Jack can now focus on other parts of his idea.

3.2.2 Beth

Beth is a graduate computer scientist who is working on a research project using Python. She has used Python before but has spent most of the past year working with other programming languages like Java. She is familiar with Object Oriented Programming but has forgotten the general syntax for Python. Her project involves data preprocessing where she might need to concatenate, filter or limit arrays of integers using standard python libraries. To get past the initial hurdle, she writes pseudo-code for how she would approach the problem in Java. She then passes it to Desc2Code and it returns the program she described with the correct syntax and appropriate methods to handle her data structures. Although the generated code might not fit exactly how she specified, the key methods are present and require few modifications to fit into her codebase. She can now use the generated code as an example for future expansions in her code, inputting further pseudo-code for more methods she can't remember the syntax of.

3.2.3 Andrew

Andrew is an experienced Software Engineer working at a medium sized company. Andrew uses Python in his day to day programming making web applications. He uses many standard libraries in his coding as well as well known community packages such as Flask (a web serving tool written in Python) as they are well documented and provide the functionality he needs. While working on a new experimental feature for his application, Andrew decides to make a sandbox environment to prototype his idea in isolation. His sandbox environment contains a skeletal Python server. Andrew requires a minimum amount of features present in his other application to effectively prototype his idea, but that code was written a long time ago and he doesn't remember the exact syntax to make the correct library calls. Despite this, Andrew has an expert grasp on what his application requires and can clearly express in English what his code should perform. He uses Desc2Code connected with an IDE plugin to enter his detailed English descriptions. These are translated into code preserving his intent in a correct syntactic form. Variables don't match exactly but Andrew is aware of the limitations of the algorithm and can perform manual modifications easily. There was no need for Andrew to leave his coding environment and put his train of thought on pause.

3.3 Functional Requirements

We outline the behaviour of the system with Functional Requirements. We prioritize these with the MoSCoW methodology Mulder (2017). This method consists in labeling each system requirement in the following order of importance: Must have, Should have, Could have, Won't have. Ordering in this fashion we have an idea of what to focus on first.

3.3.1 Must have

These are the minimum requirements that the project must satisfy to be viable.

1. Must return syntactically correct code
2. Must take a string as input
3. Must save model progress based on Hyper-parameters used
4. Must be Tensorboard compatible
5. Must run on available accelerated hardware
6. Must return comparable metrics to other sources for comparison
7. Must have a usable interface to query and receive generated code
8. Must include at least 3 different algorithms (Ex: Machine translation, lookup table + ML, Character level RNN)
9. Must output some standard benchmarks (BLEU, Precision, ROUGE...)

3.3.2 Should have

These are requirements that are not critical to the minimum specification.

1. Should return non obfuscated and understandable code
2. Should return code similar in nature to the given description
3. Should be able to infer using non accelerated hardware
4. Should be able to edit code after it is generated

3.3.3 Could have

These requirements are potential goals that improve the system but are not necessary. We include stretch goals here.

1. Could generate code through a dialogue system
2. Could use a self made synthetic dataset to train

3.3.4 Won't have

"Won't have" requirements specify features that are intentionally left out of the specification. They ensure unnecessary effort is not spent on pursuing tangential goals.

1. Will not generate helper functions not described in the description

4 | Design

4.1 Datasources

The data for our algorithms should come in a form that allows our models to learn. This means the data should show a mapping from relevant code to relevant descriptions.

We included datasources of different origins to evaluate the impact on our models and to ensure the patterns we see from our evaluation are general. Additionally, we created a separate dataset that is specifically focused on the task of code generation using data augmentation techniques to expand the set of descriptions.

4.2 Algorithms

The work in this paper is mostly focused around the algorithms used and how they work internally given the available data.

4.2.1 Retrieval

At a high level this algorithm is meant to be a simple baseline to compare our results for our other more complex implementations. The intention for it was to return already existing code that could be rated based on some similarity criteria. Since our data came in the form of parallel code description pairs, we aimed to use them as a crude translation system between the input description and the expected code output.

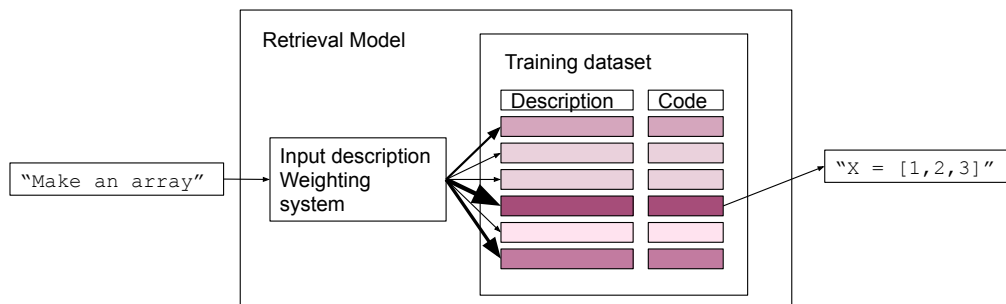


Figure 4.1: A high level diagram of the retrieval algorithm

4.2.2 Generative

Our generative model was capable of generating code from scratch by learning a mapping at a token level from input sequence to output code. It used a neural architecture similar to that used in neural machine translation, the encoder-decoder. It took in an input sequence one token at a time by encoding it in a memory state, using this state to produce the output. A notable aspect of this model was how it handled variable length input and output to produce the correct code. This model also exposed a monitoring interface during training, facilitating troubleshooting.

4.2.3 Hybrid

This model combined the capabilities of the two previous ones by enriching the input sequence to a generative model with a top retrieval code result. We see an example below.

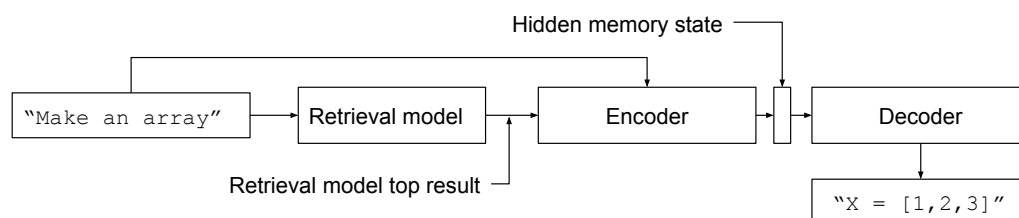


Figure 4.2: A high level diagram of the SeekNMT algorithm incorporating retrieval and generative capabilities

4.3 User Interface

We created a small user interface to act as an interface for our models in a standard programming environment. This was not the main aim for the project and was not evaluated, but would be a useful tool when displaying output.

4.4 Model comparison

We used 3 measures to quantify performance for our models.

We first used an in-domain BLEU score comparing the model's training general learning against a test set from the same dataset. This is a standard metric in research and gives a fast indication of model performance.

We then evaluated our models with BLEU an out-domain set of 55 code description pairs sourced from multiple programmers. We took a single programmer's result and use it as the ground truth while using others as a control for our metric. We then evaluated performance on this more realistic set.

We finally gathered our model's output and had programmers judge their relevance given an input description in a final user study.

5 | Implementation

5.1 Data

The data we use to train our algorithms comes in the form of code description pairs. A code description pair is a single snippet of code which could be one or multiple lines and an associated description of the code. Descriptions and code are stored in different dedicated files with each line representing a new pair.

5.1.1 Data Storage

Data science often handles vast quantities of data tens of gigabytes in size, loading these into memory can be a costly operation in terms of time and can often not even be done because of memory constraints. The solution is to not load all the data into memory at once. Tensorflow has specific libraries tailored to fast access of data in large files. The specific format used in textual data is that each line is delimited by a newline character "\n" which corresponds to a new sample of data.

Ensuring the code description pairs stay matched is paramount to successfully training any kind of model. In the case of unmatched pairs, a description will be associated with the wrong snippet of code. Any model consuming this data would result in erroneous predictions since there would be no correlation between one description and the code.

5.1.2 Description tokenization

Sentences convey their meaning through the words they contain. Before we pass any text information to a program, the text needs to be tokenised. Splitting up a string into a sequence of units is called tokenization. The individual units can be of any size. Character level tokenization for instance consists in splitting up a string into the individual characters contained therein.

In our implementation we opt for separating natural language into individual words. The decision for the granularity of the tokenization is based on the convention of the field Iyer et al. (2018); Oda et al. (2015). This is not however the only strategy available. Peters et al. (2018) show how effective language representations can be made from character level tokenization. Other kinds of encodings have shown success such as Byte Pair Encoding from Gage (1994); Sennrich et al. (2015). This last tokenisation method consists of splitting words into sub-word units having each unit carry a different meaning within the word.

To split the strings into words, we use a space separator. This is represented in code as an array of strings with each string keeping it's original order.

<pre> 1 import numpy 2 open a file 3 print the string hello world 4 get the length of a string 5 load a json file 6 save `model` to a json 7 make a for loop 8 get last element from an array 9 get tail of an array 10 check if a list is empty 11 check if `elem` is in dictionary 12 append an element to a list 13 make a dictionary with a and b as keys 14 cast to string </pre>	<pre> 1 import numpy as np 2 with open (file_path , "r") as file : 3 print ("hello world") 4 len (s) 5 json . load (file) 6 json . dump (model , file_path) 7 for i in range (10) : 8 elem_list [- 1] 9 elem_list [1 :] 10 if not elem_list : 11 if elem in elem_dict : 12 elem_list . append (elem) 13 { 'a' : 1 , 'b' : 2 } 14 str (elem) </pre>
(a) Descriptions	(b) Python code

Figure 5.1: A collection of hand made code description pairs used in the StackOverflow test. Each line corresponds to a new pair. (a) shows the descriptions that are original questions sourced from StackOverflow. (b) shows the hand written code pairing to the descriptions

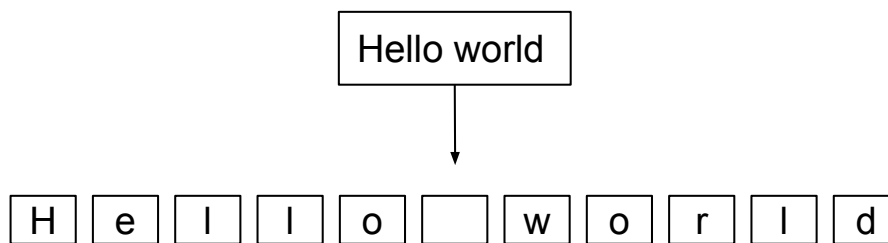


Figure 5.2: Sample character level tokenization of the string "Hello world"

5.2 Vocabulary and code tokenization

In order to pass our tokens to the algorithms we wish to use, we encode each token with an ID. This is a unique integer value that maps uniquely to the token. The collection of all unique tokens and ID pairs is our vocabulary, meaning no two tokens in our vocabulary will be the same.

Considering vocabulary size is of special importance, as we need to be able to store it in memory, finding the appropriate vocabulary size is key. The vocabulary size is intrinsically tied to the way we tokenise our data. If our tokenization is too granular the vocabulary will be small, there may be too many meanings conveyed by a single token, and the algorithms could have a hard time extracting meaningful insight.

An example of this would be character level tokenization where the entire vocabulary is made from all the letters in the alphabet and special symbols present in the corpus. The token "a", present in words like "coat", "array", "at"; carries many meanings since it is present in these words, but the character itself does not carry the individual word meanings.

When tokenizing our code we needed to keep this tradoff in mind. Code does not obey the

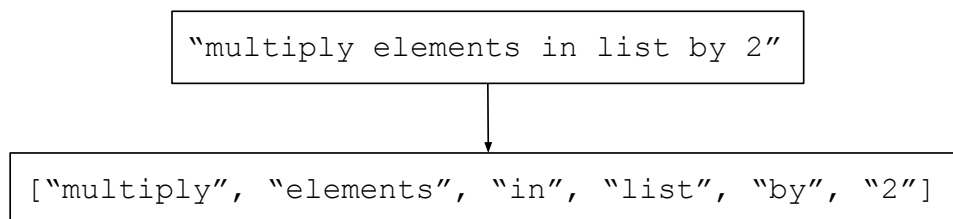


Figure 5.3: Sample space tokenization used in processing the code descriptions for the code generating algorithms

same syntax rules as English. When tokenizing with spaces this leads to an extremely large vocabulary with more than 1 Million different tokens. Spaces rarely delimit the start and end of a token. Special characters such as dots, colons, and parenthesis are often appended to the end of variable and function identifiers and these are considered unique. The better approach is to use the built-in parser from Python. This allows for a parsing of the code that is akin to its syntactic structure.

We consider also the case of new lines ("`\n`") in code. New lines are essential in Python as they delimit the end of a command in code. Our representation handles these by substituting them with a special token "DCNL". Similarly, tabs and spaces are essential for scoping statements correctly in Python. A new scope is defined when the line is indented one or more tabs away from the previous one and Python doesn't distinguish between tabs and spaces for program correctness. We substitute these for the token "DCSP".

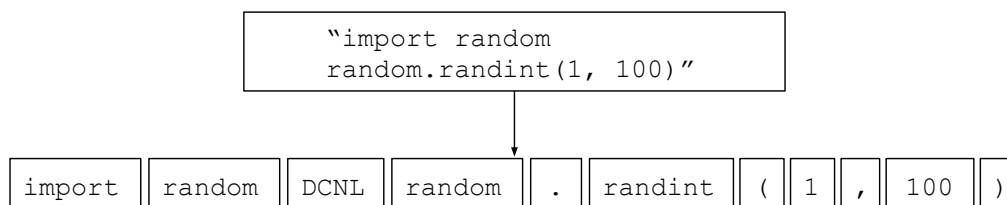


Figure 5.4: Sample code tokenization used in processing the code for the code generating algorithms

We also limit vocabulary by eliminating tokens. Deciding which tokens to eliminate is an important decision as we don't want to remove tokens that convey essential meaning. We order tokens based on their frequency and remove the least frequent terms (terms removed if they appear less than 7 times in the corpus) to comply with our memory constraints. In reducing our vocabulary size from over 1 Million to 20k (98% vocabulary reduction), we still see over 90% vocabulary coverage. That is most tokens are still present. Every vocabulary token that is removed is replaced with the special token "<UNK>" meaning "unknown".

```
{'formula':[
    ["create", "make", "instantiate", "form", "build"],
    [" a string"],
    [" containing", " holding"],
    [" 'hello'", " hello", " the following: hello"]
],
'code': ["x = 'hello'"]}
}
```

Listing 5.1: An example template when building the synthetic dataset. The formula corresponds to the different ways the description can be made. And the code indicates what the ground truth label is.

5.3 Synthetic data generation and augmentation

During the exploratory phase of the project three datasets were found; DJANGO (Oda et al. (2015), Docstring (Barone and Sennrich (2017)) and Hearthstone (Ling et al. (2016)). As none of them tackled specifically the problem of code generation for short snippets designed to reduce frequent searching, we decided to make our own dataset entirely focused on this task.

We started our dataset by researching common questions on StackOverflow. We amassed around 50 topics ranging from string operations to datetimes and time-delta operations. These topics formed the basis for our templates.

A template is an instruction set for how to construct a series of descriptions for the same label output. It consists of two parts, the formula and the code. The formula is a series of arrays that contain different formulations the description can have for the same output. This can be seen in 5.1.

We use the following code to convert a template into a series of code description pairs. This iterates for every combination of the formula into a string format and saves it into a Pandas dataframe to be later formatted into a file where each code description pair is a line in their corresponding file.

5.4 Technologies used

We used a suite of third party libraries that complement Python's original functionality. These libraries are tools often referred to in the Python scientific stack providing implementations for fast matrix multiplication, gradient calculation, and numerous evaluations metrics. The following tools were used to make our system.

5.4.1 Tensorflow

Tensorflow is an Open Source Python library for dataflow and differentiable computation built by Google noa (c). Released in 2015, Tensorflow has been rapidly adopted as one of the leading libraries for machine learning development.

We use Tensorflow to build and train our neural translation models. The library operates slightly differently to standard python operations since you transform your data by first building

a computation graph, then passing data through it and performing modifications to internal variables in the graph as the user sees fit.

This static form of programming is a characteristic feature of the library and is specifically made to facilitate the calculation of gradients from forward operations known as automatic differentiation. This allows for fast implementations of crucial algorithms in machine learning such as gradient descent to find optimal parameters to minimize functions given certain input.

5.4.2 Tensorboard

Tensorboard is a visualisation tool made to inspect graphs and plot progress from algorithms built using Tensorflow. Tensorflow integrates with Tensorboard by outputting a metadata file at specified intervals with relevant information about training losses and weight distributions. Tensorboard then takes the file and displays the data in a local web app visualizing plots, histograms and more advanced visualisations like computation graphs and interactive 3D plots.

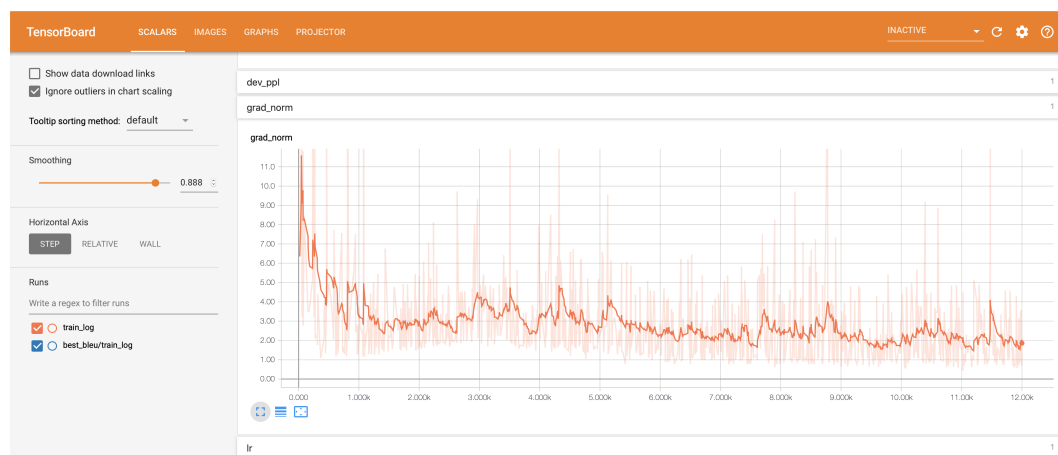


Figure 5.5: Screenshot from Tensorboard showing a gradient norm plot from the Seq2Seq model on our Django dataset. This file was downloaded from a Google Cloud instance during training and displayed locally in the browser.

5.4.3 Scikit-learn

Scikit-learn is a free software package for Python with support for many machine learning applications noa (a). The library boasts many classification, regression and clustering methods. We use this library to implement the retrieval algorithm using the Scikit-learn's TF-IDF and Cosine similarity methods.

5.4.4 Numpy

Numpy is a mathematical library that provides many tools for matrix manipulation, array operations, code vectorization and parallelisation. It integrates seamlessly into python's array structure with *ndarrays* (the datastructure at the core of the library) and allows for more efficient operations, many of which are run in compiled code for performance.

Many of the data structures that are used in this project use Numpy's *ndarray*, since it integrates not only with Python, but with Tensorflow, and Scikit-learn too.

5.4.5 Google Cloud Compute

Google Cloud compute is a subsection of Google's cloud platform. It provides users with a varied choice of virtual machines to run almost any kind of service. Cloud computing is a way to access powerful hardware useful for accelerated computation.

We make substantial use of cloud computing in this project with all models using Tensorflow benefiting from GPU parallelisation. A GPU (Graphical Processing Unit) allows for massive parallelisation of simple operations. They boast an order of magnitude more cores than traditional CPUs and are well suited for machine learning tasks as many operations can be performed independently on each core. In our applications we note massive speedups in convergence when training, making them indispensable tools for generative models. These machines are accessed through SSH (Secure SHell) to issue commands and upload data.

5.5 Algorithms

The models that we use all have a similar high level structure. They process the text data that is passed to them, they train their internal representations using this data, then they use this learned representation to infer new results given an input. Their input during training are code description pairs from separate files.

5.5.1 Retrieval

The overall intention from this model is, given a description, find the closest matching description in the training set, and return the code associated with the description.

This model is meant as a baseline system returning predictable results given an input and the available training data.

Training The inputs of this model are description code pairs. We simplify the input representation to the model by having each description be considered like a Bag of Words (BoW). This means the order in which the words in the description appear doesn't matter and that their presence is the only thing of note. Each word is attributed a score based on TF-IDF (Term Frequency over Inverse Document Frequency) from the entire set of descriptions in the training set. The description is then represented as a column vector where every row is a token in the vocabulary and the value in the row is the associated TF-IDF score associated with the word in the corpus. This has a result of representing each description as a single point in an n-dimensional vocabulary space where n is the size of the vocabulary.

We use TF-IDF vectorizer from SkLearn to transform our descriptions. This returns us an object containing all document representations.

Inference Inference is the process of obtaining a code snippet given a description. The objective for the algorithm is to output code to match as closely as possible to the intent of the description. This does not necessarily mean that the code will contain terms that match with the description.

An example for input and output would be:

Input description:

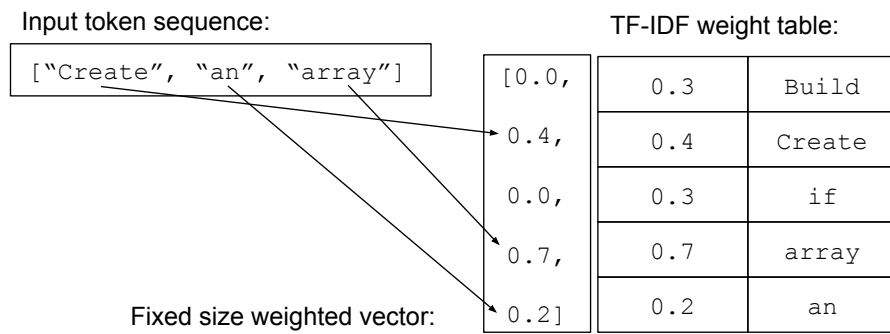


Figure 5.6: Sample process of token sequence transformation to fixed size vocabulary space vector.

Create an array with any 10 numbers

Output code:

```
np.random.rand(10)
```

Note how here the only term that is common between the description and the code is "10" yet the intent is conveyed meaningfully.

To achieve this we use the descriptions to find the most similar match. We first transform our input description into vocabulary vector space using TF-IDF weighting. This gives us a point in our n -dimensional vocabulary space along with all other descriptions. We then rank every description in the training set with the Cosine similarity measure to our input description. The Cosine similarity measure is the cosine angle at the origin given from 2 points in space. Our ranking takes the description whose vector representation makes the smallest angle with the input description. The intention being that if both descriptions have similar vocabulary, the code from one will be relevant to the other.

5.5.2 Generative

We approached the task of Code Generation from a generative neural translation standpoint. At a high level, neural translation uses the encoder-decoder architecture and involves reading the input sentence, transforming it into a fixed size vector, then using that vector to decode the meaning into the target medium. Architectures like these have lately gained popularity in tasks such as text translation Cho et al. (2014a;b), text summarizing Iyer et al. (2016) and question answering Yin et al. (2015).

Implementing this model was a main focus throughout the project, with many iterations as challenges were overcome. In the following sections we cover the details of how we adapted this flexible architecture to our problem.

We used the canonical version of the encoder-decoder architecture for our first attempt at neural translation. This was the one described in the inspiring work from Cho et al. (2014b). The input is represented as a sequence of tokens of arbitrary length. The variable nature of the input is one of the defining features of this network. It is key because descriptions need not follow a fixed pattern to perform optimally.

Input description: `get` `the` `absolute` `path` `of` the `__file__`, `repo_dir` is name of `the` directory two levels above `it`.

Result: `get` `the` unicode representation of `the` `absolute` `path` `of` `base`, substitute `it` for `base_path`.

```
base_path = abspath ( base )
```

Ground Truth:

```
repo_dir = os . path . dirname ( os . path . dirname ( os . path . abspath ( __file__ ) ) )
```

Figure 5.7: A visualisation of matching words between an input description and a similar training sample from the retrieval model. We note also the associated code with the training sample and the ground truth below.

Before a token can register with the encoder, it is passed to an embedding layer. A word embedding is a mapping from a 1-dimensional word ID (obtained through the transformation of vocabulary into integer values) to an n-dimensional vector representation. This is a way of capturing the semantics of a word in a way that is more attainable for our models to understand. This sequence of embeddings is passed as the input to the encoder.

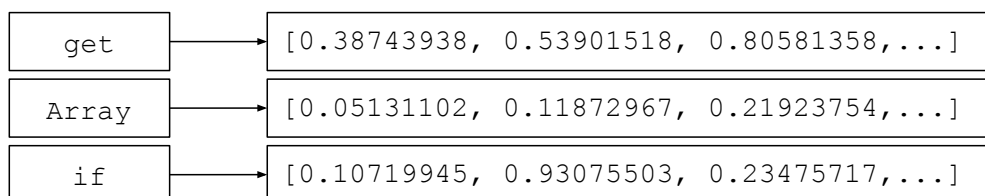


Figure 5.8: Word embedding example mapping to n-dimensional vector space. All embeddings are of equal size.

The encoder reads this sequence one vector at a time. The input is processed and the meaning is encoded into the hidden state. We used a stack of LSTM cells to iterate over each embedding one time step at a time. At the final time step we kept the encoder's hidden state. We interpreted this vector as the meaning of the input description encoded in a latent space. The size of the memory is a hyperparameter that has a direct impact on the model capabilities, and is subject to the same issues of over and under-fitting that plagues other machine learning systems.

Following this, we fed the final memory state to the decoder, another LSTM cell stack. A starting token "<START>" is given to the decoder to jump start the decoding process. This last token is also embedded into the same space as other vocabulary terms. The cell's output is passed to a fully connected layer and transformed into a probability distribution over every possible token. This process yields the first token in the output sequence. We continued the decoding process by

feeding the latest generated token as the input to the decoder. This kind of decoding process is called Greedy Decoding since we take the most likely token generated at each time-step. We stop once a specific "<END>" token is produced indicating the end of the decoding process. Other methods exist such as Beam Search decoding (Wu et al. (2016)) where the top n token predictions (where n is the beam width) are kept for the next stage, and a constant beam of n possible translations are kept.

We obtained a quality measure for the error (loss) in our sequence predictions by using a sequence cross entropy. This measure takes in the ground truth sequence (the associated code pair in our dataset) and gives a score by how close the two sequences match. Cross entropy is a measure for how likely the ground truth is sampled from the predicted distribution. This gets applied to the sequence of tokens and their corresponding match.

In practice all the above operations were performed on multiple samples from the training set. This is called a batch and is a core concept in modern machine learning since it can help converge faster (Smith and Le (2017)) and increase computation speed thanks to parallelization. We used batching in our network by grouping every sequence of tokens in our batch together. They were then fed into the encoder and processed by the decoder all in parallel for each time step.

Input token sequences from our descriptions of course don't always match in length. We tackled this problem by padding our sequences with a special "<PAD>" token. Semantically, this token holds no information and is only used to give every sequence in a batch the same length. This also occurs during the decoding process where a generated output code token sequence could be shorter than another in the same batch. We see below an example of batch padding.

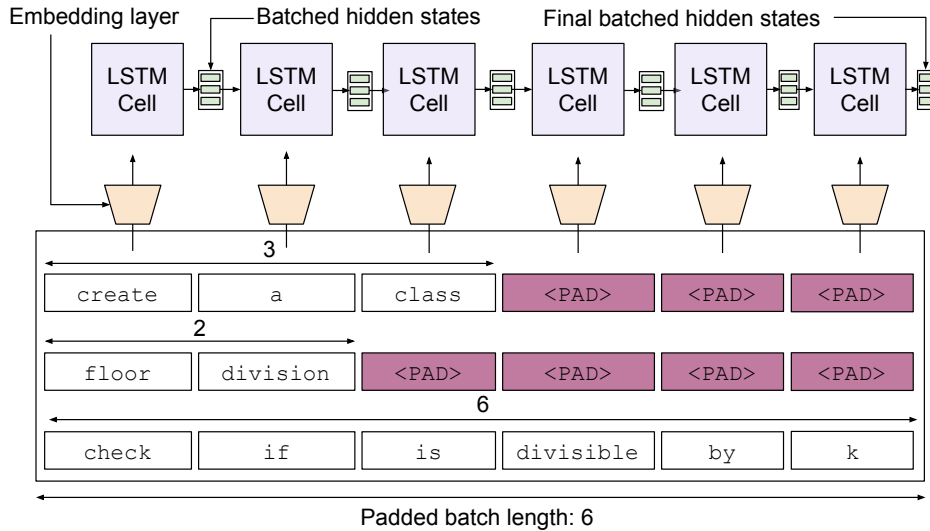


Figure 5.9: Example of a padded batch of size 3 being fed into an LSTM encoder. Note how the hidden states keep the same size as the batch.

We added an attention mechanism to the decoder to give it more information when generating the output token sequence. This is in line with current baseline models for code generation Iyer et al. (2018); Rabinovich et al. (2017). Attention is a method described in the inspiring paper from Bahdanau et al. (2014), in it they describe a technique to align input hidden states in the different steps in the decoding process. This compliments our previous technique by not only passing the final hidden state to initialize the decoder, but every single hidden state made during the encoding process at each time step. The attention to each hidden state is given by a layer that weights each hidden state based on it's relevance to the output. Attention weights will tend to

shift towards the end of the input as the output also ends. This is because the network learns that certain words in the input sequence are relevant at certain time steps.

Issues during implementation While implementing this attention architecture in Tensorflow, we ran into training issues where the model did not converge. This resulted in the use of a similar implementation by Google on GitHub Luong et al. (2017). This implementation converges effectively and takes advantage of all accelerated hardware available from the Google Cloud platform.

The input format for this system is almost identical to our previous handmade implementation, the only difference being the tokenization strategy being always spaces rather than an a custom method necessary for effective separation of code tokens. We solved this by preprocessing the tokens from the code samples to be first isolated by the code tokenizer, then put together and saved to a new file with spaces added where needed.

5.5.3 Hybrid: SeekNMT

We hope to improve on these last models by bringing forward a new model to attempt to address the shortcomings of the two previous ones, the idea being to integrate our retrieval model results into the encoding stage of the generative model. We brand this: Seeking Neural Machine Translation (SeekNMT). In the sections below we discuss the architecture, inference and training cycles.

Architecture During the implementation of the previous models. We noticed how each model's output has certain characteristics. The retrieval model having every code description pair presented during training will return a properly formatted code snippet. This is a very desirable trait for a model to have since the user might not be very familiar with the generated code, and as such will trust the code to behave as the description suggests. We expanded the original input description with this new code snippet by concatenating it to the token sequence with a separator token "<END_RETRIEVAL>" to help the model know when the code snippet has ended and the description begins. If it's wasn't there, the model (and particularly the attention mechanism) could confuse the input code to extend into the description. The addition of the separator token actually changes the problem definition since by having the encoder deal with two input sequences. They are meant to convey different meaning and thus help address the task of code generation in different ways. The code section should impact the kind of structure needed in the code while the description should clarify the methods to be used in more detail.

We see this technique of splitting up different kinds of inputs in other works in machine translation and specifically code generation from Iyer et al. (2018). Here context such as methods, parameter types, headings, and return types are included with the natural language descriptions and are fed into an NMT system.

We then passed this improved description token sequence to the generative translation model. We first ran these through the embedding model, now shared to accommodate the additional tokens and decode using the weighted hidden states from attention.

Training Training occurs in two phases, we first build a training set consisting of the original descriptions paired with retrieved code snippets. To achieve this we first train our retrieval model with the full training set. We then pass the descriptions from the training set again and take the second best predictions from the model and append them to the descriptions. The reason for taking the second best retrieval result is because the trained retrieval model will return the exact code snippet for the description because it has been trained with it. What we aim to do is to train the model with an approximate guess. So by taking out the best retrieved code snippet which would yield a perfect cosine similarity score of 0 we achieve this result.

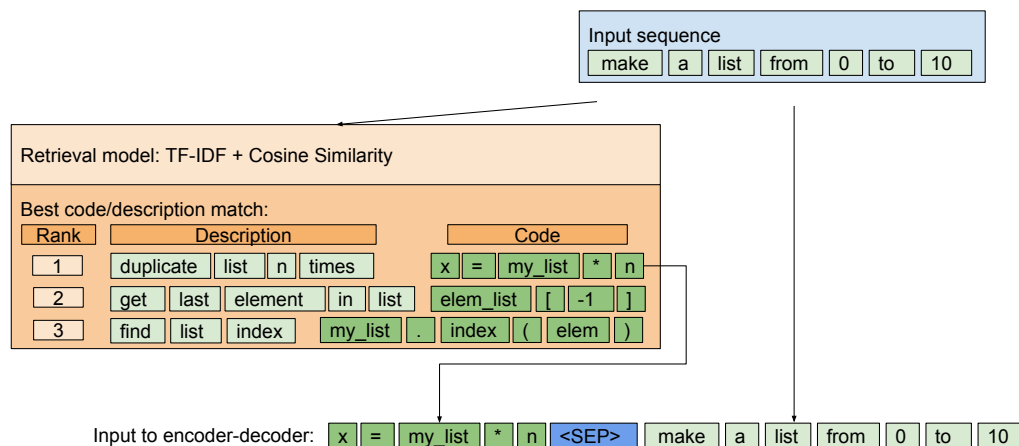


Figure 5.10: Visualisation of the retrieval phase during inference from the SeekNMT algorithm. The best retrieved code snippet is concatenated to the original description with a separator token. This makes up the new input to the generative model.

```
if __name__ == '__main__': DCNL <END_RETRIEVAL> given name is script

print ( sorted ( LIST_0 ) ) <END_RETRIEVAL> reorganize LIST_0 ascending and
      output it to the console

sorted ( LIST_0 ) <END_RETRIEVAL> organize VARIABLE_0 ascending then empty LIST_0
```

Listing 5.2: A series of input sequences to SeekNMT. We note how before the separator token the code retrieved is a good estimate of the requirements of the description but not always accurate such as in line 3

For large training sets computing the cosine similarity table is a costly operation since we are creating an $N \times N$ matrix where N is the size of the dataset, so we batch the creation of the augmented samples to not overwhelm our constrained memory.

Next we saved our descriptions with the second best code prediction with the separator token "<END_RETRIEVAL>". The translation model then took the code description pairs (the descriptions have code embedded in them but they are treated the same) and tokenizes with spaces, substitutes infrequent tokens for "<UNK>" and converts to IDs.

5.6 Metrics

We cover here any implementation detail for our methods of evaluation and metrics of our models.

5.6.1 BLEU

Our first implementation of BLEU used scikit-learn's translation module which contains an implementation of the metric. Our first attempts during the earlier stages of the project used

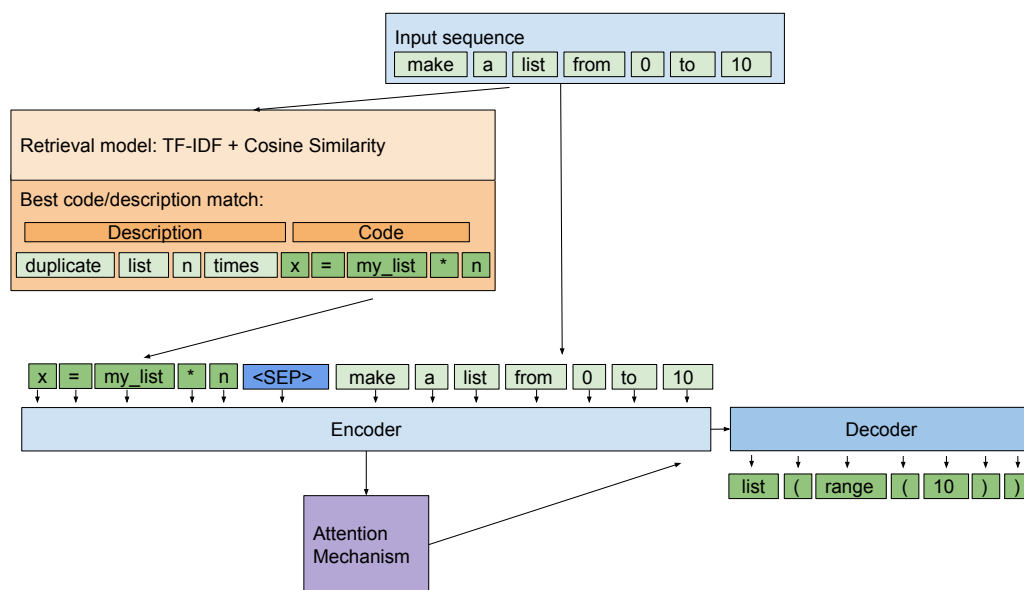


Figure 5.11: Visualisation of dataflow in the SeekNMT architecture during inference.

BLEU with a positive weight for only unigrams. This had the same effect as unigram precision described in the metrics section of the background. We swiftly abandoned this though when we attempted to match the results of translation models in the related research Iyer et al. (2018). Unigram precision tended to inflate results by basing the score only on the presence of tokens in both the reference and predicted token sequence, as opposed to taking order into account.

When integrating the implementation of the seq2seq model with attention from Google (Luong et al. (2017)) into our system, we noted how it came with an implementation of BLEU. By adapting this code to generically take in two files with parallel sentences we used their version to remove any implementation errors on our part that could interfere with the results. All our results for BLEU are computed using this script which weights up to n-grams of size 4 with a weighting scheme of (1.0,1.0,1.0,1.0) where each float in the list corresponds to the weight given to the presence of that n-gram in the token sequence.

5.6.2 User Study

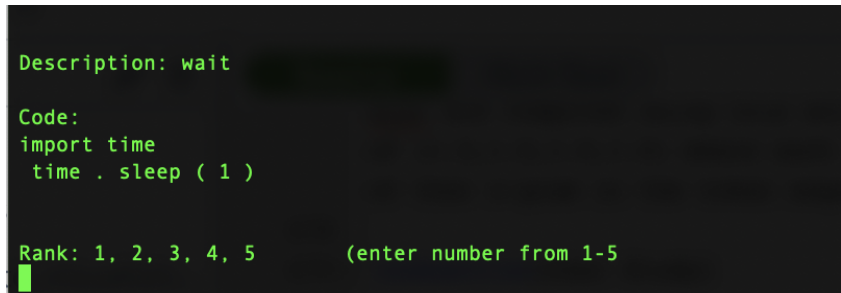
During the course of this project we employed two user oriented studies.

The first study was intended to gather code from a group of programmers as controls for our out-of-domain evaluation. There are few implementation details to cover and as such the utility and execution will be covered in the following evaluation chapter.

Our second study consisted of asking users experienced in programming to rate how useful they found the generated code output from our models given a description from our out-of-domain test set. This user study was originally intended to be carried out on paper with only 6 descriptions to rate per algorithm/dataset combination (12 combinations in total) and 2 human controls for a total of 84 ratings. Handing around 15 pages to users.

The shift to 55 questions for a total of 770 questions lead to an amount of paper that was daunting for any user or myself to handle. As such, a way of running the questions using a command

line interface was made to streamline the process of rating for users. The outputs from every model for the out-of-domain descriptions were placed into a public GitHub project along with a Python script to format all 770 questions to the users.



```

Description: wait

Code:
import time
time . sleep ( 1 )

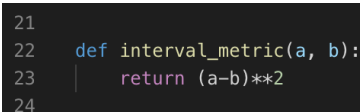
Rank: 1, 2, 3, 4, 5      (enter number from 1-5

```

Figure 5.12: User view of the snippet ranking user study. This view presents the user with the description passed to the code generating process as well as the code and the ranking scale. Once the number was input through the keyboard, the view moved on to the next question.

To make the experience of rating so many code snippets more pleasant an automatic save state was created at the start of execution of the study. This save state contained a list of answered and unanswered questions in a JSON format. This file was created at the same level as the execution file and was searched for by the script to restore a previous line of questioning rather than create a new save state. The save state file was updated every time a user rated a snippet to not require any additional input from the user to save.

When creating the save state file, a random ID is assigned to the state to identify between users results. Results for each user are then presented using Google Sheets, a tabulation program hosted in the browser. Here we are able to compute the binarization of the ratings by use of cell functions. When the binary relevance results are obtained. We use a script to calculate the relevance using Krippendorff's alpha. The interval metric is used which consists in taking the distance between users ratings. We then measure statistical significance between the binary results using T-test.



```

21
22     def interval_metric(a, b):
23         return (a-b)**2
24

```

Figure 5.13: Python function depicting the interval metric used in the calculation of Krippendorff's alpha. The interval metric returns a value proportional to the distance between the ratings of two participants.

5.7 User and Program Interfaces

The data formats for the algorithms we use was very specific and not optimal for users when coding. An infrastructure surrounding our system to have users interface with the models we create was necessary.

5.7.1 Server

We decided to interact with our algorithms in a simple way by using the same methods used for training and inference, files. The server takes the data field and stores it into a file with

each description taking a new line. This file is saved and a system command is made to run the algorithms in an inference mode. The result is then saved to a separate file and picked up by the server and returned.

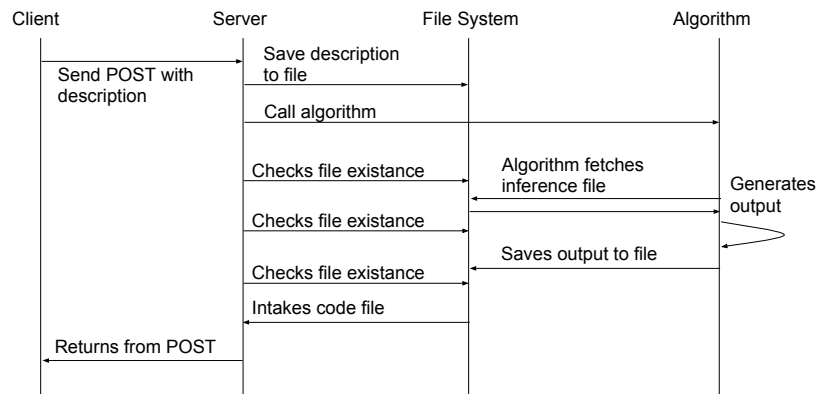


Figure 5.14: Detailed server flow diagram showing path from request from client to returned code.

5.7.2 User Interfaces

The user interfaces for our system needed to be able to take in a user's description in an efficient way that is non obtrusive when programming. This means having the interface being directly integrated into the development environment. The choice for our environments was conditioned on the target audience for our system. Given that our target programming language was python we opted for popular python editors.

Jupyter Notebook Jupyter notebook stands out for certain reasons. It is primarily aimed at fast prototyping with code being executed in small sections on a web interface. It is not the final destination for code as it behaves much like a notepad that you run once and thus has little risk of interfering with live systems. This makes it the ideal candidate for our interface.

Our extension to the platform consists of a code cell that is run to enable functionality. The cell overrides some JavaScript functionality by tapping into some standard functions from Jupyter like adding keyboard shortcuts, and modifying cell contents.

A user can type their description inside their code enclosed in a delimiting character sequence "@@". The extension can then be triggered by pressing the key combination Ctrl+Shift+h. The extension will pick up the enclosed description and substitute it with the code. The code, once inserted, is selected to give the user a clear indication for what was generated.



Figure 5.15: An example code snippet generated by the retrieval model from the django dataset using the Jupyter notebook client interface.

6 | Evaluation

In this section we cover how we evaluated our system. This consisted of running experiments that attempted to answer our initial hypothesis and research questions.

6.1 Datasets

We started with the datasets we wished to use. The kind of data we feed to our models is critical to the quality of the output of our system.

In this project we used the same format of dataset for similar problems in machine translation. Datasets of this format consist of source and target parallel pairs, often kept in separate files. In following the same convention we searched for datasets that adhere to the same format.

The datasets researched during this project with the purpose of code generation are presented below. We describe their properties and compare their suitability for the task.

6.1.1 Django

The Django dataset stems from work from Oda et al. (2015). In their paper they present the task of generating pseudo-code from code. This is the inverse problem from the one we face. Their task is to take in a line of code and create annotations to help with high level understanding. It consists of 18805 parallel pairs from a line by line segmentation of the entire Django source code. The associated descriptions are line by line annotations of this source code written by one software engineer.

```

3  define the __init__ method with argument self .
4  substitute cache for value under the alias key of self . _caches . caches dictionary .
5  define the method __setattr__ with self , name and value as arguments .
6  delete the attribute name from the cache dictionary under the DEFAULT_CACHE_ALIAS key .
7  call the DefaultCacheProxy function , substitute it for cache .
8  from __future__ import unicode_literals into default name space .
9  import module time .

```

(a) Django dataset single line annotations

```

3  def __init__ ( self ) :
4  self . _caches . caches [ alias ] = cache
5  def __setattr__ ( self , name , value ) :
6  return setattr ( caches [ DEFAULT_CACHE_ALIAS ] , name )
7  cache = DefaultCacheProxy ( )
8  from __future__ import unicode_literals
9  import time

```

(b) Django Python code

Figure 6.1: A sample of parallel code description pairs from the Django dataset. This particular selection is from the test set for the evaluation consisting of pairs unseen to the model during training.

We note, however, some shortcomings. Most obviously, the annotations and code are broken up in a line by line manner. This kind of separation does not lend itself well to code since a single intention can span multiple lines such as functions, if statements and loops. The annotations do not take account of the context and are thus poor depictions of the code. This is mostly due to the nature of the task that is being attempted since simplifying a detailed implementation is rougher than trying to generate code for the associated annotation. Secondly, the annotations are written by one software engineer biasing the entire dataset to a particular style of description.

6.1.2 Docstring

This dataset is the product of work from Barone and Sennrich (2017). In their paper they argue that the current available code description pair datasets are not sufficient in size and diversity for current attempts in neural translation. They state that available datasets, Django included, are too narrowly focused on specific domains and that a more varied dataset representing programming in a larger scope is required.

Their solution is to source their code description pairs from top Python repositories from GitHub, a web-based hosting service for version control using Git. Projects are broken up into the functions they contain with the commented function description (or docstring) considered the description and the associated internal call being the code. This split is more faithful to the nature of code by taking into account new lines and indentation. New line characters are translated to the token "DCNL" and an indentation mark in Python which could be either a Tab character or 2 or 4 spaces which is marked as "DCSP". This allows the entire description and code pair to be saved in one line in a file.

Shortcomings of the dataset are that it is not properly filtered. Many pairs are very poor with code being inserted into the description and vice versa. This makes the dataset very noisy. Given its size of 100k pairs, it is not possible to check the quality manually. In addition, some code snippets consist of over 160k tokens. Upon closer inspection of these long examples, it is clear that many snippets of code are automatically generated and contain too much detail to be accurately described.

```
Obtain the OS version from the
show version output
Print output to STDOUT
```

Listing (6.1) Docstring for Python function

```
with open (
    '../show_version.txt' ) as
show_ver_file :
    show_ver = show_ver_file .
        read ( )
    print obtain_os_version (
        show_ver )
```

Listing (6.2) Python function body

Figure 6.2: A sample of parallel code description pairs from the Django dataset. This particular selection is from the test set for the evaluation consisting of pairs unseen to the model during training.

6.1.3 CoNaLa

Code from Natural Language (CoNaLa) is a dataset made by Yin et al. (2018) with the intention of sourcing high quality concise snippets from StackOverflow. Their approach involves mining the site for questions that are of the "how to" format. They use a classifier trained on a hand

crafted set of over 2k code description pairs to identify high quality snippets. They manage to scrape the site for over 600k description code pairs. However we find their mined dataset to be excessively noisy for our application as we see below.

```
How can I plot hysteresis in matplotlib?

How to make curvilinear plots in matplotlib

is it possible to plot timelines with matplotlib?
```

Listing (6.3) StackOverflow questions

```
plt.show()

plt.show()

plt.show()
```

Listing (6.4) Python snippets

Figure 6.3: A subset of code description pairs from the CoNaLa mined dataset. Note how all the code snippets are lacking when capturing the intent of the descriptions by simply showing the last step of a plotting task using the matplotlib Python library.

In this project we use the 2k+ handwritten pairs which are of much higher quality.

6.1.4 Synthetic

The intention behind our dataset was to have a varied set of descriptions mapping to similar code snippets. Making a dataset with enough variability in the samples by hand is challenging given that other datasets in the field of machine translation are very large with the WMT14 English-German dataset (noa (b)) capturing 4.5M pairs being considered medium in size.

We attempt to remedy this by augmenting our data through parallel ways of describing a code snippet. In doing so we hope to create a dataset that can provide general language patterns in a narrow domain of function calls.

We started by exploring StackOverflow questions for Python. This served as a start for the initial templates which have no expansion terms. We then expand our amassed questions using the template driven augmentation (seen in implementation) into a dataset with over 50k code description pairs. These pairs consist of hand written expansion terms aimed to maximize sentence coherence and correctness.

```
1321 log find the start value LIST_0
1322 take the start item LIST_0 then return
1323 take 1st value of LIST_0 after that VARIABLE_0 in the next seconds the value of time
1324 get the first value of LIST_0 print it
1325 take first element LIST_0 and print
```

(a) Synthetic descriptions

```
1321 print ( LIST_0 [ 0 ] )
1322 return LIST_0 [ 0 ]
1323 result = LIST_0 [ 0 ] DCNL VARIABLE_0 + datetime . timedelta ( seconds = result )
1324 print ( LIST_0 [ 0 ] )
1325 print ( LIST_0 [ 0 ] )
```

(b) Synthetic Python code

Figure 6.4: This is a sample of the training set from the Synthetic data augmentation

6.1.5 Dataset comparisons

Despite our datasets being focused on the same task of code generation, they exhibit very different characteristics at fundamental levels. Most obviously they vary greatly in size which is an important fact when training neural systems since generalization often comes from a large variety of samples.

The kind of code being described is also a differentiating factor Docstrings are often formatted differently from syntax queries for sites like Google and StackOverflow. Docstrings and annotations, as in the Django dataset, possess a style focused on describing code after it's written rather than before.

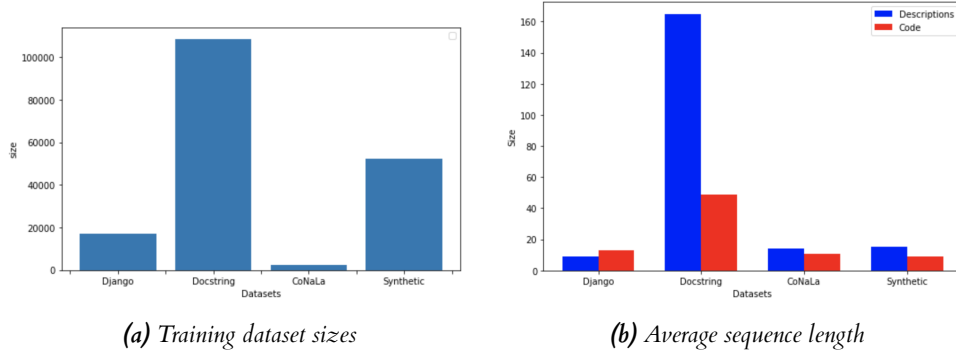


Figure 6.5: Property comparisons for our different datasets.

6.2 Testing

Assessing if machine learning systems are learning at all is a challenging task. In the next sections we cover how we verified convergence of our models to desired results.

6.2.1 Dummy prototype dataset

In order to prototype models quickly and effectively, we made a very small test set simulating input descriptions and associated code. The intention of this set was to be extremely easy for the model to find a distinguishing pattern between the input and the output sequences. We were looking for the model to overfit to the data with ease.

Our prototyping dataset consisted of 8 training samples seen below and 2 for our test set.

We found our models were able to quickly converge to incredibly high scores of near 100% within a minute of training. This test set proved useful on multiple occasions troubleshooting hyperparameters and unsuccessful models.

6.2.2 ConCode

ConCode is a dataset for Java code description pairs stemming from work from (Iyer et al. (2018)). Their paper is strongly aligned with this project as they use NMT techniques to synthesize code from Java function docstrings (java-docs). A defining feature of this dataset is how their

1 hello foo	1 hello foo = hello foo
2 hello bar	2 hello bar = hello bar
3 array foo	3 array foo = array foo
4 array bar	4 array bar = array bar
5 hello foo baz	5 hello foo baz = hello foo baz
6 hello bar baz	6 hello bar baz = hello bar baz
7 array foo baz	7 array foo baz = array foo baz
8 array bar baz	8 array bar baz = array bar baz

(a) Prototype train descriptions (b) Prototype train code

1 hello baz	1 hello baz = hello baz
2 array baz	2 array baz = array baz

(c) Prototype test descriptions (d) Prototype test code

Figure 6.6: The entire prototype training and testing set used to check model convergence and training for fast iteration.

descriptions contain contextual information from the class the code is derived from. This has a similar intention to SeekNMT by feeding information that could be relevant when decoding.

Although we didn't use this dataset for our evaluation, it served as inspiration for our input as it uses special separators that are useful indicators for the attention mechanism. It also served to confirm the validity of our models by replicating their results.

We downloaded their dataset which comes with an accompanying script to create a train, development and test set. Our testing includes the retrieval and neural translation models. This is because their paper includes those same models using similar techniques such as TF-IDF, cosine similarity, and NMT with attention. Our intention was to roughly match their results with the same data they used.

Model	dev BLEU	test BLEU
Our Retrieval	18.3	21.1
ConCode Retrieval	18.15	20.27
Our NMT	19.6	23.2
ConCode NMT	21.0	23.51

Figure 6.7: Table showing a comparison of our models trained and evaluated with the ConCode dataset and the baselines from Iyer et al. (2018). Their numbers originate from their paper.

We see that both models results are very close suggesting our algorithms were capable of reproducing results and thus able to create meaningful statistics.

6.3 Training

In this section we cover the training phase of our models and their parameters.

6.3.1 Retrieval

Training for the retrieval models is fast with Scikit-learn providing efficient implementations that are able to run locally on a laptop. The training phase consisted in building the vector representations of all descriptions. Training times varied from a few seconds to 5 minutes. This

swift training time is a very desirable quality for a model that might need to adapt to new snippets of code fast.

The model did not present many hyperparameters for tuning as it is quite simple. Our only initial point of contention was the vocabulary constraints, however we later found that to be a non issue so we used a full vocabulary for every dataset.

6.3.2 NMT and SeekNMT

The neural translation models are not as straight forward as retrieval when it comes to training. We were able to train both locally since Tensorflow has a CPU runtime. This was useful for testing if the models were able to initialize correctly without producing errors. However, the hardware soon became overwhelmed with training times often exceeding 20 days.

We drastically improved our training time by using GPU accelerated instances from the Google Cloud platform. This platform provides a variety of instance configurations. During the course of this project we used many instances in parallel with each one running a separate algorithm-dataset configuration. Our instance of choice was an 8 core instance with 26Gb RAM and an Nvidia Tesla K80 GPU at 0.2£/h. We also tried similar instances with more powerful GPUs, but the speedup did not justify the 6x jump in cost at 1.2£/h. The K80 instance provides sufficient computation acceleration with the GPU showing acceptable usage hovering around 60-70% utilisation.

```

+-----+
| NVIDIA-SMI 410.72      Driver Version: 410.72      CUDA Version: 10.0      |
+-----+
| GPU   Name           Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|  Memory-Usage | GPU-Util  Compute M. |
+-----+
|  0   Tesla K80          Off      | 00000000:00:04:0 Off |                    |
| N/A   46C    P0      95W / 149W | 10903MiB / 11441MiB |      62%      Default |
+-----+

+-----+
| Processes:                      GPU Memory |
| GPU       PID    Type    Process name      Usage      |
+-----+
|  0        21012   C       python3           10886MiB |
+-----+

```

Figure 6.8: Terminal screenshot during the training session for the NMT model on the Docstring dataset. The terminal is showing GPU stats from the `nvidia-smi` command.

During training we monitored the instance's statistics with commands such as "top" and "nvidia-smi" providing valuable information when models don't perform optimally or are overwhelming the hardware.

Statistics specific to the model can be seen using Tensorboard. During training we could download locally a metadata file generated by the algorithm containing all training information. We then viewed the content using Tensorboard. Statistics such as training graphs, time, losses, training and testing BLEU scores are all shown.

Hyperparameters Both NMT and SeekNMT posses numerous hyperparameters available for tunning. However, we restricted ourselves to a particular set given that they were popular from similar research from Iyer et al. (2018). As such there was minimal tuning, and any was mostly done manually for testing. Our parameters for all NMT based models were as follows: `-num_train_steps=12000 -steps_per_stats=100 -num_layers=2 -num_units=128 -dropout=0.2 -metrics=bleu -batch_size=256 -infer_batch_size=256 -src_max_len=2000 -tgt_max_len=150 -src_max_len_infer=200 -tgt_max_len_infer=150 -attention=scaled_luong`

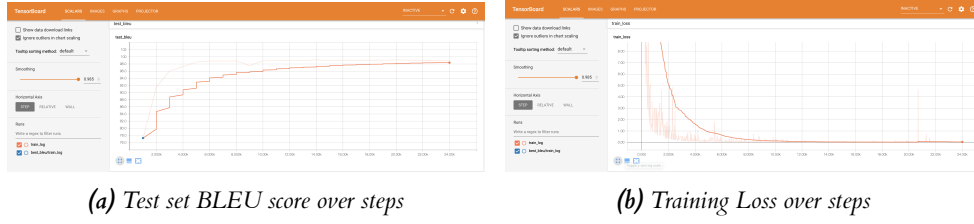


Figure 6.9: A Tensorboard visualisation once training has concluded for SeekNMT on our Synthetic dataset.

6.4 Metrics

In order to measure how well our models perform at the task of code generation we used the following metrics.

6.4.1 In-domain: Dev BLEU

In-domain testing consists of evaluating a model based on samples that are separate but derive from the training set for the model. This is a development metric to be used to tweak the performance of the model when changing the hyper-parameters.

We used BLEU as our evaluation metric in this domain. BLEU is popular in many machine translation tasks and shows co-relation between it's scores and human judgments in many cases Wu et al. (2016). BLEU works on a per-token basis, and we tokenize our output sequences using a space separator.

Our test set was made by taking the totality of the samples available from every dataset and splitting them into a training and testing set randomly at a rough ratio of 90% to 10%.

6.4.2 Out-domain: Stack BLEU

As Ben-David et al. (2010) notes, testing sets can perform well when originating from a set of samples removed from the model's training set. Out of domain testing consists in drawing samples for testing that are separate from the training set and thus might lead to results that are closer to those expected in the real world. The assumption is that both sets will be drawn from different distributions showing traits of generalization.

To achieve this we made a new set of 55 descriptions. These were still short concise descriptions about code inspired in StackOverflow questions. Our goal was to have a set of ground truth labels that were sourced separately from the training sets and used BLEU to score the model results against these different labels. We justified the choice of 55 descriptions to make evaluation for statistical significance possible. The original set contained only 6 descriptions.

We sourced code for these descriptions from 2 human annotators. These annotators were familiar with Python and had experience programming. We framed this as a user study and briefed them of all the experiment details such as their anonymity, the focus of the study and goal. The participants were intended to write code they would normally write given such a description. Usual tools like online searches (Google, StackOverflow, etc.) and examples were allowed in code sourcing. This was to make the code associated with the descriptions as real to a user as possible.

Our reasoning for the two parallel code samples was to have one for the ground truth labels to be compared against for all other models, and the other as a human control to highlight the differences

between the code generated by our algorithms and the natural process of programming. We named the use of this dataset in the rest of this paper: Stack; from StackOverflow.

6.4.3 Manual Relevance Assessment: Precision@1

To assess the real-world effectiveness, we collected manual relevance judgments. We formulated this as a second user study where each annotator, an undergraduate CS student, assessed the relevance for the 55 code descriptions from the Stack dataset.

We formulated this study as a series of questions asking the user to rate the relevance of a generated snippet of code against a ground truth label from a human annotator given a description on a scale from 1 to 5 where 1 is "unusable" and 5 is "exact translation".

Description: Reverse list `x`

Django Retrieval

```
parts.reverse ( )
```

Rank: [Irrelevant code] 1_2_3_4_5 [Exact translation]

Figure 6.10: A single rating question showing the description passed to the algorithm, the generated code from said algorithm and the human ground truth. Note, this was not the final version given to users as it still contains the names of the algorithms and datasets used. And it also is still using the paper format.

A single question is composed of a description, a generated code snippet for the description and a reminder of the rating scale. We evaluated the generated code for our 55 Stack descriptions on our 3 algorithms: retrieval, NMT, and SeekNMT. We also gathered the results for each model after training of 4 datasets: Django, Docstring, CoNaLa, and Synthetic. This resulted in 12 different algorithm-dataset combinations and 2 additional human generated code snippets from the previous user study as control results. This gave 14 snippets per description for a total of 770 code description relevance judgments. We also kept any relevant comments from this study regarding the quality of the generated code.

All participants used the command line interface described in the implementation section when rating the snippets. The interface briefs them on the intention of the study, their anonymity, and an example for the kind of rating that should be given in an example.

The study concluded once they completed all 770 ratings and the participants returned the results in a JSON file with their anonymous ID. The result being a table with a rating for each algorithm-dataset pair across all 55 descriptions. We measured user agreement by comparing the results from every user using Krippendorff's alpha, Krippendorff (2011). This measure of agreement evaluates the reliability with which participants can rate a description with the same value. It can take values from 0 to 1 with results above 0.7 considered reliable.

Different measures for this value exist, with the main distinguishing factor being the distance measure between ratings. For our agreement we used a distance measure proportional to the distance between the rating of two users, this is Krippendorff's interval metric. We achieved an agreement measure of 0.712.

We aimed to obtain a measure of performance for our models suggesting the relevance the snippets they produce. To achieve this we computed the precision at 1 (prec@1) which means taking the top result from our models and rating their relevance. We joined the results from our

different participants into a single rating by taking a rating when all users agree on the rating and breaking disagreements randomly. We binarized the rating results from our study by setting a threshold of relevance at the value of 3 (inclusive). All above were deemed relevant by users, all below were not. We then plotted the results by taking the percentage of relevant results per algorithm-dataset pair.

6.5 Results

In this section we examine the results of our experiments on the various collections and methods. We start with baseline retrieval and neural translation models trained on various corpora and evaluated using BLEU. We then focus on results of our new proposed Retrieval+NMT model. Finally, we use the manual assessments as a way to measure the real-world effectiveness of the methods and we contrast this with BLEU-based evaluation.

Description:

execute a command 'command' in the terminal from a python script

Human 1:

```
import subprocess
subprocess.Popen(command)
```

Human 2:

```
os.system(command)
```

Retrieval CoNaLa:

```
os.system('dir c:\\')
```

NMT CoNaLa:

```
os.delete('some')
```

Retrieval+NMT CoNaLa:

```
os.system(<unk> <unk>)
```

6.5.1 Baseline Retrieval and NMT results

The section below presents results of the retrieval and NMT models trained on a variety of corpora. We also compare to a human coder baseline.

Model	Dev BLEU	Stack BLEU
Retrieval Django	45.9	4.3
Retrieval Docstring	21.8	1.2
Retrieval CoNaLa	11.8	5.1
Retrieval Synthetic	75.6	9.7
NMT Django	51.2	3.4
NMT Docstring	1.7	0.3
NMT CoNaLa	16.3	2.3
NMT Synthetic	92.3	2.2
Human	/	36.5

Figure 6.11: In-domain (Dev) and Test (StackOverflow) scores on multiple datasets.

In-domain sets are common in the field of code generation to benchmark the effectiveness of models. We find Neural Translation outperforms Retrieval for most datasets for in-domain sets. However, our StackOverflow set is much more difficult, shown by the vastly lower scores, and yields the opposite results as the retrieval method finds more relevant code than NMT by a significant margin. Humans show vastly improved results, though the fact that they don't score near 100 BLEU points suggests BLEU is limited. Results like these show the difficulty for a model to perform well in real world situations.

Dev BLEU	NMT	Retrieval+NMT	Delta
Django	51.2	53.5	+4.5%
Docstring	1.7	1.9	+11.8%
CoNaLa	16.3	13.0	-20.2%
Synthetic	92.3	98.0	+6.1%
Stack BLEU			
Django	3.4	4.6	+34.3%
Docstring	0.3	1.7	+466.6%
CoNaLa	2.3	3.8	+65.2%
Synthetic	2.2	5.5	+150.0%

Figure 6.12: BLEU Score comparison of NMT and Retrieval+NMT algorithms on Dev and Stack sets

Adding retrieval capabilities to Neural Translation yields improvements in almost every dataset. Most notable improvements are found in the Stack BLEU scores pushing marginally past our previous benchmarks. We theorize passing the actual description with the retrieval model's best guess allows the hybrid translation model to estimate appropriate syntax and target length while changing relevant tokens to match the description. We note a strong dependency on datasets for scores with the new model finding similar difficulty generalizing on the Docstring dataset as opposed to more focused datasets like Django, Conala and our Synthetic one.

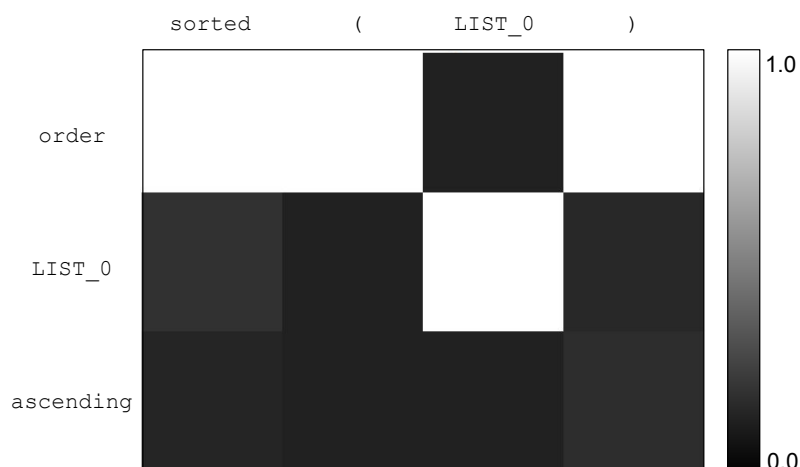


Figure 6.13: Example of an attention weighting from an input code description to the output code token sequence. Higher weights indicate a larger weighting of the input hidden states. Note how the higher activations for the input token "order" for the whole output sequence might indicate overfitting in paying excessive attention to characteristic tokens in the training set.

However, looking closely at results from our synthetic dataset, we notice poor generalization

qualities for both NMT and SeekNMT, with most outputs being very similar to samples from the training set. Looking closer at Figure 6.13 we see that our attention model is not making much notice of important words like "ascending" showing signs of overfitting.

6.5.2 Manual assessment

We now compare Stack BLEU results to human rated effectiveness.

Model	Stack BLEU	Prec@1
Retrieval Django	4.3	20.9%
Retrieval Docstring	1.2	18.2%
Retrieval CoNaLa	5.1	20.9%
Retrieval Synthetic	9.7	15.5%
NMT Django	3.4	6.4%
NMT Docstring	0.3	4.6%
NMT CoNaLa	2.3	3.6%
NMT Synthetic	2.2	4.6%
Retrieval+NMT Django	4.6	0.9%
Retrieval+NMT Docstring	1.7	3.6%
Retrieval+NMT CoNaLa	3.8	4.5%
Retrieval+NMT Synthetic	5.5	7.3%
Human	36.5	90.0%

Figure 6.14: Stack BLEU & Precision@1 for all models on their StackOverflow descriptions

As participants rate the effectiveness of the generated code for all 55 descriptions, we observe statistically significant improvements (>95% confidence) for retrieval over other methods. Taking a closer look at the outputs we notice the translation models, despite picking up keywords in the description, often void the output by inserting irrelevant tokens that make the outputs unsuitable to users. Human coders manage very high relevance despite differences in the tokens used (highlighted by lower BLEU score). Since BLEU gives a score based on n-gram similarity, these findings reinforce the view that other metrics akin to the polymorphic nature of code will highlight better success in a model's predictions. Participants reported having difficulty distinguishing between similarly bad predictions, suggesting user ratings show most telling results when the generated code is near high quality. Below is an example of a result with a high BLEU score, but is marked as non-relevant. It shows that BLEU does not capture the redundancy well. And that translation models pick up on important details like "numpy", while retrieval models may miss key aspects, but return well-formed code.

Description:

```
import numpy
```

Retrieval:

```
import code
```

SeekNMT:

```
return numpy . array ( numpy . array
( numpy . array ( numpy . array (
```

7 | Conclusion

7.1 Summary

In this dissertation we tackle the challenging task of code generation from natural language descriptions. We formulate the problem as a translation task where an input sentence describing a programming intention needs to be mapped to a concise snippet of code. We approach the task with three distinct algorithms. The first, a retrieval system ranking code snippets based on the similarity of matching descriptions to the input. Second, a neural machine translation baseline incorporating state-of-the-art techniques such as attention. Lastly, we integrate the two previous approaches into a new model (SeekNMT) designed to improve the input to the translation model by retrieving the most similar code snippet and combining it with the original description. We pair this with a user interface integrated into a programmer's development environment.

We gather multiple standard datasets for training, and make additional synthetic dataset using template driven augmentation. Our investigation covers the different algorithms performance using multiple metrics from in and out-domain test sets. We also ask programmers to rate generated snippets for relevance given an input description. Our results shows a slight improvements in standard benchmarks for SeekNMT over other models and retrieval being preferred by programmers by a statistically significant margin.

7.2 Contributions

- New model for code generation integrating retrieval and neural translation
- An new dataset of code description pairs focused on code generation
- We show how retrieval models have desirable output features not present in translation models

7.3 Reflection

This project has taught me a great deal on the difficulties of making any kind of contribution to a field of academia. The project was researched focused and had an emphasis on testing for rigor. During the development of the algorithms, notably the baseline NMT system, I learned that building an algorithm capable of matching other researchers in the field in accuracy performance and maintainability is something extremely difficult and a task I underestimated.

Making meaningful contributions is even harder, requiring a deep understanding of the complexities of the problem being faced. Had I started the project equipped with the knowledge I have now, I would emphasize having a clear vision for the desired outcomes when making a system. The focus for this project changed over the course of it's lifetime and ultimately left gaps of unused research that were not pertinent to the outcome.

However, the project was successful. With meaningful results making one small step for the field of code generation.

7.4 Future Work

Programmers constantly rethink approaches and re-factor their programs as they jot down new code. New ideas spark as the code is changed and enhanced. Retaining this kind of creativity is crucial for good software. The approach we take here is an iterative and collaborative way to generate code between an agent and a programmer, similar to the Software Engineering (SE) technique of Pair Programming.

In our future work we propose to break the task of programming into multiple roles where the user and the agent code a single solution to a problem by splitting the task of programming into a "driver" who writes the implementation and a "navigator" who watches over and looks out for the design as a whole while questioning the implementation.

The interactions would be modeled as a complex dialogue with many utterances relying on already existing knowledge of previous statements or code. This would push the boundaries of what is currently possible with current conversational AI systems. An important note is that the techniques regarding common goal setting, planning, and collaborative learning are not specific to the task of code generation and could provide a benefit to the field of conversational AI as a whole.

A | Appendices

5. No information about the evaluation or materials was intentionally withheld from the participants.
Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.
6. No participant was under the age of 16.
Parental consent is required for participants under the age of 16.
7. No participant has an impairment that may limit their understanding or communication.
Additional consent is required for participants with impairments.
8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.
9. All participants were informed that they could withdraw at any time.
All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.
10. All participants have been informed of my contact details.
All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.
11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation.
12. All the data collected from the participants is stored in an anonymous form.
All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title Pose 2 code : snippet generation
 Student's Name Carla Gemmell
 Student's Registration Number 2209560
 Student's Signature [Signature]
 Supervisor's Signature [Signature]
 Date _____

Figure A.1: Signature from supervisor for 55 snippet generation user study

```

def generatePermutations(data, formula, code):
    index = [0 for arr in formula]
    index[0] = -1
    fullIndex = [len(arr)-1 for arr in formula]
    permutations = []
    while(index != fullIndex):
        for i in range(len(index)):
            if(index[i] < fullIndex[i]):
                index[i] += 1
                index[:i] = [0] * i
                break
        perm = ""
        for i in range(len(formula)):
            perm += str(formula[i][index[i]])
        permutations.append({'description':perm, 'code':code})
    print(len(permutations))
    return data.append(pd.DataFrame.from_dict(permutations), ignore_index=True)

```

***Listing A.1:** This is the function that drives most of the data generation by combining all of the separate formulations of the descriptions into code description pairs*

5. No information about the evaluation or materials was intentionally withheld from the participants.
Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.
6. No participant was under the age of 16.
Parental consent is required for participants under the age of 16.
7. No participant has an impairment that may limit their understanding or communication.
Additional consent is required for participants with impairments.
8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.
9. All participants were informed that they could withdraw at any time.
All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.
10. All participants have been informed of my contact details.
All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.
11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation.
12. All the data collected from the participants is stored in an anonymous form.
All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title Desc 2 code: snippetrank
 Student's Name Carlos Gemmell
 Student's Registration Number 2209560
 Student's Signature [Signature]
 Supervisor's Signature [Signature]
 Date _____

Figure A.2: Signature from supervisor for Snippet rating user study


```

import numpy
open a file
print the string hello world
get the length of a string
load a json file
save 'model' to a json
make a for loop
get last element from an array
get tail of an array
check if a list is empty
check if 'elem' is in dictionary
append an element to a list
make a dictionary with a and b as keys
cast to string
create dictionary from list of tuples
get tomorrow datetime
delete an item with key 'key' from 'mydict'
iterate through 2 arrays at the same time
make a list from 0 to 10
run main if script is run from terminal
duplicate list n times
multiply elements in list by 2
get unique elements in list 'my_list'
return list where elements are greater than x
calculate execution time
create a class
floor division
check if is divisible by k
does list contain
index of element in list
binary to string
one line for loop
shuffle array
get random int
declare global variable
print new line
check if string is number
raise exception
get dictionary keys as list
check type
maximum of list
wait
declare enum
add key value to dictionary
get subarray from elements a to b
convert string list to string
return empty dictionary
first character in string
stack pop
convert dictionary to json
Reverse list 'x'
check if string 'string' starts with a number
Trimming \n from string 'myString'
execute a command 'command' in the terminal from a python script
print numbers in list 'list' with precision of 3 decimal places

```

Listing A.2: Input descriptions for the Stack dataset

```

f = np . array ( ( x . blabla ( 1 ) ) . reshape ( 1
exec ( file . blabla )
print ( '\xd0\xbf\xd1\x80\xd0\xb8' . path . is_file ( ) )
str os . getcwd ( )
json . sleep ( array )
foo = [ str ( i ) for i in range ( 100 ) ]
float ( 1 )
print ( max ( itertools . isnan ( a , - 1 ) ) )
alist [ 0 ]
if ( not seq ) : DCNL DCSP DCSP pass
( 7 in d )
[ '' for ( i , x ) in mylog ]
u = urllib . request . items ( )
str <unk> <unk> . len ( )
[ ( x + tuple ( y ) for y in b ] for y in range
now = datetime . datetime . now ( )
try : DCNL DCSP DCSP try : DCNL DCSP DCSP DCSP DCSP print ( 'i' in [ 0
print ( ( listone . setdefault ( key , lambda x : x . count ( 1 )
list ( itertools . fromkeys ( list2d ) )
os . Popen ( [ 'rm' , '-r' , 'some.file' ] )
list ( map ( my_dictionary . fromkeys , keys ) )
[ ( i * dict2 ) for x , y in zip ( functions , values ) ]
print ( list ( itertools . product ( list ( range ( 10 ) ) , repeat =
[ ( x in list1 if x not in b ]
datetime . datetime . strftime ( '2007-03-04T21:08:12' , '%Y-%m-%dT%H:%M:%S' )
( 'key1' , nargs = False )
redirect ( 'Home.views.index' )
( not a ) . exists ( )
result = numpy . array ( )
[ [ ] for i in range ( 3 ) ]
str ( myString )
del lst [ : ]
np . zeros ( ( 6 , dtype . '<U2' ) ) DCNL print ( parsed_html . transpose ( r ) )
random . random . e ( 'url' )
variable = [ ]
print ( "{}{}" . format ( date ) )
if some_string : DCNL DCSP DCSP pass
canvas . set_position ( 'firstname' )
random_choice = list ( a )
os . getcwd ( )
[ [ ] for i in range ( 3 ) ]

variable = [ ]
( key , dict )
random_choice = sys . stdin . getall ( 0 )
a [ : int ( x ) ]
type ( x )
"" . join ( reversed ( string ) )
. translate ( 'name' , default = 'application/json' )
r = requests . put ( )
[ [ ] for n in x ]
s . replace ( ',' , '' )
re . strip ( name , '/' )
os . system ( <unk> <unk> )
<unk> <unk> . join ( lst )

```

Listing A.3: Results from NMT model trained on the CoNaLa dataset on the Stack dataset

```

globals ( ) . update ( importlib . import_module ( 'some.package' ) . __dict__ )
os . system ( 'start "$file" )
hello world' [ : : ( - 1 ) ]
return '' . join ( random . choice ( string . lowercase ) for i in range ( length
    ) )
json . loads ( open ( 'sample.json' ) . read ( ) . decode ( 'utf-8-sig' ) )
urllib . request . urlretrieve ( 'http://search.twitter.com/search.json?q=hi' ,
    'hi.json' )
for fn in os . listdir ( '.' ) : DCNL DCSP DCSP if os . path . isfile ( fn ) :
    DCNL DCSP DCSP DCSP DCSP pass
del a [ ( - 1 ) ]
x [ : - 1 ] + ( x [ 1 : ] - x [ : - 1 ] ) / 2
if ( not a ) : DCNL DCSP DCSP pass
( 7 in a )
c . extend ( a )
list ( d . items ( ) )
ME' + str ( i )
dict ( x [ 1 : ] for x in reversed ( myListOfTuples ) )
datetime . datetime . now ( ) . isoformat ( )
mydict . pop ( 'key' , None )
[ [ sum ( item ) for item in zip ( * items ) ] for items in zip ( * data ) ]
myList = [ i for i in range ( 10 ) ]
os . system ( 'gnome-terminal -e \'bash -c "sudo apt-get update; exec bash"\'' )
list ( dict ( ( x [ 0 ] , x ) for x in L ) . values ( ) )
( a . T * b ) . T
set ( [ 'a' , 'b' , 'c' , 'd' ] )
[ i for i in range ( len ( a ) ) if a [ i ] > 2 ]
sys . exit ( )
getattr ( test , a_string )

( 7 in a )
"This DCSP DCSP is a DCSP DCSP string" . split ( )
del a [ index ]
np . fromstring ( '\x00\x00\x80?\x00\x00\x00\x00\x00\x00\x00\x80' , dtype =
    '>f4' )
for fn in os . listdir ( '.' ) : DCNL DCSP DCSP if os . path . isfile ( fn ) :
    DCNL DCSP DCSP DCSP DCSP pass
np . random . shuffle ( np . transpose ( r ) )
ord ( 'a' )
variable = [ ]
a \n b \r\n c ' . split ( '\n' )
b . isdigit ( )
raise ValueError ( 'A very specific bad thing happened' )
list ( d . items ( ) )
isinstance ( s , str )
len ( max ( i , key = len ) )
p . wait ( )
my_list = [ ]
data . update ( { 'a' : 1 , } )
[ x for x in a if x not in b ]
str ( i )
lst [ : ] = [ ]
newcontents = contents . replace ( 'a' , 'e' ) . replace ( 's' , '3' )
pd . concat ( [ GOOG , AAPL ] , keys = [ 'GOOG' , 'AAPL' ] , axis = 1 )
json . load ( u )
s [ : : ( - 1 ) ]
b . isdigit ( )
myString .rstrip ( '\n\t' )
os . system ( 'dir c:\\\'' )
[ ( '%.2d' % i ) for i in range ( 16 ) ]

```

Listing A.4: Results from Retrieval model trained on the CoNaLa dataset on the Stack dataset

```

. realpath ( '/' )
with open ( 'C:/name/MyDocuments/numbers' , 'r' ) as f : DCNL DCSP DCSP if (
    blabla in line
isdigit ( )
len ( my_string )
stream : DCNL DCSP DCSP DCSP DCSP pass
stream = 0 : DCNL DCSP DCSP DCSP DCSP print ( exc )
img . append ( 0 )
newlist in list ( a . values ( ) )
alist [ : - 1 ]
if ( not j ) : DCNL DCSP DCSP pass
( 'key1' in lst )
del dict ( d )
ord ( 0 )
\x02\x08\x93'
[ d == [ ] for d in items ]
sys . version_info
my_list . append ( a )
self . & ( self . request . urlretrieve ( self . urlretrieve ( self . POST )
list ( x )
app . run ( 2010 , self . CRITICAL )
[ : ]
[ j ]
any ( d )
set ( d )
datetime . datetime . strptime ( days = 0 )
. Thread ( 'ascii' , 'ignore' )
stream : 10 }
#ERROR!
for ( d2 , d2 ) in result )
del [ 1 , 5 ] ]
hex ( number )
mylist . append ( 1 )
list1
array ( )
my_list [ 2 ]
myString . encode ( 'ascii' , 'ignore' )
if ( conv is i ) : DCNL DCSP DCSP pass
urllib . request . urlopen ( 'http://www.stackoverflow.com' )
set ( len ( a ) )
] . 5
& ( session )
. dic [ : , ( None ) ]
[ 'C' ] == 1
new_list = 0 == 0
word = { 'size' : len ( x )
hex ( s )
all ( i )
ord ( <unk> is <unk> )
. date ( )
stream ]
x [ : ]
for line in lst . readlines ( ) : DCNL DCSP DCSP pass
line ) DCNL except ValueError : DCNL DCSP DCSP pass
os . delete ( 'some' )
print ( unsorted_list . count ( 0 ) )

```

Listing A.5: Results from SeekNMT model trained on the CoNaLa dataset on the Stack dataset

7 | Bibliography

- scikit-learn: machine learning in Python – scikit-learn 0.20.3 documentation, a. URL <https://scikit-learn.org/stable/>.
- The Stanford Natural Language Processing Group, b. URL <https://nlp.stanford.edu/projects/nmt/>.
- TensorFlow, c. URL <https://www.tensorflow.org/>.
- Keras LSTM tutorial - How to easily build a powerful deep learning language model, Feb. 2018. URL <https://adventuresinmachinelearning.com/keras-lstm-tutorial/>.
- D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473 [cs, stat]*, Sept. 2014. URL <http://arxiv.org/abs/1409.0473>. arXiv: 1409.0473.
- A. V. M. Barone and R. Sennrich. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv:1707.02275 [cs]*, July 2017. URL <http://arxiv.org/abs/1707.02275>. arXiv: 1707.02275.
- S. Ben-David, J. Blitzer, K. Crammer, A. Kulesza, F. Pereira, and J. W. Vaughan. A theory of learning from different domains. *Machine Learning*, 79(1-2):151–175, May 2010. ISSN 0885-6125, 1573-0565. doi: 10.1007/s10994-009-5152-4. URL <http://link.springer.com/10.1007/s10994-009-5152-4>.
- J. Brownlee. Encoder-Decoder Recurrent Neural Network Models for Neural Machine Translation, Dec. 2017. URL <https://machinelearningmastery.com/encoder-decoder-recurrent-neural-network-models-neural-machine-translation/>.
- C. Callison-Burch, M. Osborne, and P. Koehn. Re-evaluation the Role of Bleu in Machine Translation Research. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL <http://www.aclweb.org/anthology/E06-1032>.
- K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv:1409.1259 [cs, stat]*, Sept. 2014a. URL <http://arxiv.org/abs/1409.1259>. arXiv: 1409.1259.
- K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv:1406.1078 [cs, stat]*, June 2014b. URL <http://arxiv.org/abs/1406.1078>. arXiv: 1406.1078.
- P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, 1994.
- D. Hiemstra. A probabilistic justification for using tfx idf term weighting in information retrieval. *International Journal on Digital Libraries*, 3(2):131–139, 2000.

- S. Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, Apr. 1998. ISSN 0218-4885, 1793-6411. doi: 10.1142/S0218488598000094. URL <http://www.worldscientific.com/doi/abs/10.1142/S0218488598000094>.
- S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1195. URL <http://aclweb.org/anthology/P16-1195>.
- S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Mapping Language to Code in Programmatic Context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium, Nov. 2018. Association for Computational Linguistics. URL <http://www.aclweb.org/anthology/D18-1192>.
- J. J. Jeng and B. H. C. Cheng. Using formal methods to construct a software component library. In G. Goos, J. Hartmanis, I. Sommerville, and M. Paul, editors, *Software Engineering – ESEC ’93*, volume 717, pages 397–417. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993. ISBN 978-3-540-57209-1 978-3-540-47972-7. doi: 10.1007/3-540-57209-0_27. URL http://link.springer.com/10.1007/3-540-57209-0_27.
- K. Krippendorff. Computing Krippendorff’s Alpha-Reliability. page 12, Jan. 2011.
- W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. KoÄNiskÄ, F. Wang, and A. Senior. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, Berlin, Germany, Aug. 2016. Association for Computational Linguistics. URL <http://www.aclweb.org/anthology/P16-1057>.
- C. Liu, Y. Wang, K. Kumar, and Y. Gong. Investigations on speaker adaptation of LSTM RNN models for speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5020–5024, Mar. 2016. doi: 10.1109/ICASSP.2016.7472633.
- M. Luong, E. Brevdo, and R. Zhao. Neural machine translation (seq2seq) tutorial. <https://github.com/tensorflow/nmt>, 2017.
- G. Mishne and M. D. Rijke. Source Code Retrieval using Conceptual Similarity. In *Proc. 2004 Conf. Computer Assisted Information Retrieval (RIA04)*, pages 539–554, 2004.
- P. Mulder. MoSCoW Method: for setting Requirements by order of Priority, Aug. 2017. URL <https://www.toolshero.com/project-management/moscow-method/>.
- Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, Lincoln, NE, USA, Nov. 2015. IEEE. ISBN 978-1-5090-0025-8. doi: 10.1109/ASE.2015.36. URL <http://ieeexplore.ieee.org/document/7372045/>.
- C. Olah. Understanding LSTM Networks, Aug. 2015. URL <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>.

- K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL <http://www.aclweb.org/anthology/P02-1040>.
- M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. *arXiv:1802.05365 [cs]*, Feb. 2018. URL <http://arxiv.org/abs/1802.05365>. arXiv: 1802.05365.
- M. Rabinovich, M. Stern, and D. Klein. Abstract Syntax Networks for Code Generation and Semantic Parsing. *arXiv:1704.07535 [cs, stat]*, Apr. 2017. URL <http://arxiv.org/abs/1704.07535>. arXiv: 1704.07535.
- Rumelhart David E., Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. Oct. 1986. URL <http://www.cs.toronto.edu/~hinton/absps/naturebp.pdf>.
- M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, Nov. 1997. ISSN 1053-587X. doi: 10.1109/78.650093.
- R. Sennrich, B. Haddow, and A. Birch. Neural Machine Translation of Rare Words with Subword Units. *arXiv:1508.07909 [cs]*, Aug. 2015. URL <http://arxiv.org/abs/1508.07909>. arXiv: 1508.07909.
- R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 905, Shanghai, China, 2006. ACM Press. ISBN 978-1-59593-375-1. doi: 10.1145/1134285.1134448. URL <http://portal.acm.org/citation.cfm?doid=1134285.1134448>.
- S. L. Smith and Q. V. Le. A Bayesian Perspective on Generalization and Stochastic Gradient Descent. *arXiv:1710.06451 [cs, stat]*, Oct. 2017. URL <http://arxiv.org/abs/1710.06451>. arXiv: 1710.06451.
- I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014. URL <http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>.
- P. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct. 1990. ISSN 00189219. doi: 10.1109/5.58337. URL <http://ieeexplore.ieee.org/document/58337/>.
- Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Å. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv:1609.08144 [cs]*, Sept. 2016. URL <http://arxiv.org/abs/1609.08144>. arXiv: 1609.08144.
- J. Yin, X. Jiang, Z. Lu, L. Shang, H. Li, and X. Li. Neural Generative Question Answering. *arXiv:1512.01337 [cs]*, Dec. 2015. URL <http://arxiv.org/abs/1512.01337>. arXiv: 1512.01337.

- P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. *arXiv:1805.08949 [cs]*, May 2018. URL <http://arxiv.org/abs/1805.08949>. arXiv: 1805.08949.
- A. Zhao, L. Qi, J. Dong, and H. Yu. Dual channel lstm based multi-feature extraction in gait for diagnosis of neurodegenerative diseases. *Knowledge-Based Systems*, 145:91–97, 2018.