Automatic Documentation of Fine-Grained Elements in Source Code

Eric Hambro [1]

MSc Machine Learning

Matko Bošnjak & Prof Sebastian Riedel

Submission date: September 2018

Abstract

Code documentation is the natural language component of code, designed to help humans in code comprehension. It is a vital part of well-written code. Documenting fine-grained elements of code, such as individual arguments, is a laborious manual task prone to error, inaccuracy or neglect. Automating this task would be valuable to the software engineering industry, while providing a stepping stone to research tasks such as type inference. However, automatic generation of such descriptions has so far not been attempted in the literature, in part due to a lack of an appropriate dataset.

In this paper we present a novel dataset sourced from most popular open-source Python libraries, to investigate the automatic documentation of function arguments from their source code. In particular we investigate generation from two modalities of the underlying code: the names in function signature, and the structure of the abstract syntax tree. We find that by analysing names as a sequence of characters, we are able to generate plausible descriptions from combinations of argument names, function names and co-argument names, by both rote learning and sequence-to-sequence models. We also find that by representing our abstract syntax tree as a sequence of paths, we are able generate descriptions using an original architecture, our Code2Vec Decoder. We demonstrate that this model surpasses a comparable rote learner, even under partial or full occlusion of lexical data from the syntax tree.

We demonstrate that the combination of these modalities yields an improvement on each of the individual models, suggesting that multimodal models may be a promising future direction of research, and finally we investigate the challenge of applying our trained model to code from different libraries.

Acknowledgements

There are a number of people I need to thank for their help and support throughout this project: first and foremost, Matko Bošnjak, for his tireless patience and guidance; secondly Sebastian Riedel for his thoughtful advice and interest in the project; also friends, family and the rest of UCL Machine Reading.

Also, thanks to the Whiney Beaches, for making it fun.

Contents

Chapter 1

Introduction

In this chapter we motivate our investigation into the documentation generation of fine-grained elements of code, present the central problem in question, and formalise the objectives of this investigation and report. Finally we present an overview of the contributions of our research and the following structure of the thesis.

1.1 Motivation

Source code is the purest medium through which humans communicate their instructions to computers. Yet these instructions need to be comprehensible to other humans as well. Understanding source code is already a valuable skill in industry today and one that is likely to grow in importance, as it is vital for software development and maintenance of code. Work that facilitates human understanding of source code is therefore highly likely to find value in software development, engineering, research, and beyond.

Source code documentation is the natural language component of code, designed to provide clear communication to human readers. It is often an abstract summary of the purpose or function of the code, though sometimes an overview of the underlying subroutine. Such summaries prove valuable in their own right, but are only one aid in understanding the source code they relate to. Often an understanding of the individual, fine-grained elements of the source code, such as arguments or variables, requires investigation by the reader - or an explicit mention by the documentation writer.

With the advent of dynamically-typed languages, explanations of individual elements of source code, such as arguments, have renewed importance. These languages do not give explicit type information, such as ‘string’ or ‘integer’, and can lead to critical run-time errors when arguments are misused. In fact, a number of open-source projects[1] and industrial company guidelines[2] now require an explicit description of the arguments of each function in all contributions to their projects. The constant enumeration of these is manual, tedious, and prone to error, inaccuracy (through code churn) or even neglect.

As a result an algorithm that could automatically provide human descriptions of arguments would find great use in software tooling, either to help code readers understand undocumented code, or to help code writers automate their task. Such an algorithm would also be a stepping stone to providing full descriptions of all elements in the source code, further helping humans with code comprehension. Given the advances that industry hope to make in the fields of automatic program synthesis, such tools will be vital to help humans understand a computer’s output. Lastly such an algorithm would be a stepping stone to the solving problem of type inference in dynamically-typed languages, which would help software developers prevent critical run-time failures in large scale software.

1.2 Code & The Naturalness Hypothesis

Historically, academic research into software tooling has focused on formal and deductive methods. For instance, in the field of bug finding, static analysis has formed the basis of methods to reduce runtime errors of software, either through explicit checks (Cousot et al., 2007), or through reasoning about sets of execution traces (Bessey et al., 2010). This research exploits the fact that code has a structure that is logically consistent and can be reasoned about mathematically. As a result, these deductive methods have been

[1]https://numpydoc.readthedocs.io/en/latest/format.html#sections

[2]https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings

attractive due to their formal nature and the guarantees they can offer in the worst case.

However, the proliferation of online open-source code (Dyer et al., 2013) has provided opportunities for a more statistical approach. Instead of formalising rule-based systems, the development of tools can now be guided by the most statistically significant features of the code-in-the-wild. Such a transition, from rule-based approach to statistical approach, signals a different end-goal for these tools, abandoning worst case guarantees for better performance on average (Allamanis et al., 2017a). This promises to be of much value to the growing computing industry. It also opens up tooling research to the wealth of statistical machine learning techniques.

This transition is fundamentally justifed under the Naturalness Hypothesis, proposed by Allamanis et al. (2017a). This hypothesis suggests that *"Software is a form of human communications; software corpora have a similar statistical properties to natural language copora; and these properties can be exploited to build better software tools".*(Allamanis et al., 2017a) The hypothesis draws inspiration from Don Knuth's idea of *literate programming*(Knuth, 1984), that the primary task of programming should be "explaining to human beings what we want a computer to do..."(Knuth, 1984), not simply preparing a set of commands for a machine.

Allamanis cites the successes that recent statistical natural language processing (NLP) approaches have had when applied to 'Big Code', dating back to Hindle et al. (2012), to support his hypothesis. He also points to evidence in cognitive science studies, that demonstrate that brain behaviour in reading programming languages is akin to reading "natural languages with greater expertise"(Floyd et al., 2017).

We mention the Naturalness Hypothesis because it is a fundamental inspiration for the approach and methods of our research. We believe that it is only in looking at the statistical 'naturalness' properties of both source code and the language accompanying it, that generalisable, extenisble progress will be made in human-oriented tasks such as code summarization, code explanation, and documentation generation.

1.3 Problem Formulation

As the field of statistical research into code and its 'naturalness' properties is relatively new, we feel there are opportunities to investigate the field of documentation generation, as it relates to specific elements of code. To this end, we examine the automatic generation of brief descriptions or summaries of Python function arguments, by examining their code, contextual data, and descriptions found in docstrings.

In particular we wish to investigate:

- whether reasonable descriptions can be generated from just lexical names in the function signature?
- whether reasonable descriptions can be generated from the function's abstract syntax tree, without the lexical data?
- whether these models, can be combined in a way that surpasses each individually?
- whether such models can work both in an 'in-project' setting and an 'out-project setting'?

1.4 Our Contributions

Our contributions include a new and challenging dataset suitable for investigation in machine summarization of source code elements, comprised of 40,000 Python arguments and descriptions. To the best of our knowledge, this is the first dataset suitable for the task of argument description generation, and the first study of such a problem. We demonstrate the informative power of function signatures, using both Rote Learner models, and neural translation architectures. We also present a new architecture capable of generating argument description solely from a syntax tree, and demonstrate its effectiveness in this task. We demonstrate its ability to learn, despite partial and full occlusion of lexical data, and also present a multimodal archtitecture combining the lexical and syntactical code features that surpasses both individual models. We finally demonstrate the difficulty of applying such models between datasets, and show there is a still a significant challenge remaining on this dataset.

1.5 Structure of the Thesis

In addressing the research questions posed above, we first present a background of theory in Chapter 2, and related work in Chapter 3. Chapter 4 gives in depth presentation of our new dataset, along with an analysis of its composition, while Chapter 5 presents the method and models which we used to investigate our central research question. Chapter 6 presents a report of our experiments, their results, and their analysis. Finally we present a conclusion of the results in Chapter 7, with an elaboration on potential future work.

Chapter 2

Background

In this section we present an overview of the theoretical items necessary to our task of generating natural language from code features. We start with an overview of language models, defining what they are, and noting where our task fits in the traditional NLP tasks of conditional langauge generation. Then we provide an overview of the relevant representation learning techniques we use to achieve our goal. Finally we conclude with a brief overview of the structure of source code, in particular its representation as an Abstract Syntax Tree, and set of Path-Contexts.

2.1 Language Modelling

2.1.1 Introduction

Although natural language has the potential to be rich and complex, more often than not it is mundane and repetetive (C. E. Shannon, 1951). It is such repetitiveness that makes it possible to predict in many instances. For instance, in natural English the phrase "*The music is loud, turn it ...*", is much more likely to be followed by '*off*', or '*down*', than the word '*around*'. This is because the actual distribution of utterances is very sparse in the vast space of possible utterances in natural language.

In our task, we seek to generate sequences of natural language, $\{y_1, y_2, ..., y_n\}$ from a set of features of code, $\mathcal{C}$. Ideally we seek to do this through maximising a probability:

$$p(y_1, y_2, ..., y_n|\mathcal{C}) \propto p(\mathcal{C}|y_1, y_2, ..., y_n)p(y_1, y_2, ..., y_n) \tag{2.1}$$

where the proportionality follows from Bayes' rule. This $p(y_1, y_2, ..., y_n)$ term in this equation represents the probability of the sequence occuring naturally in the language. This term is what language models aim to represent.

Language models are an unsupervised means of determining the natural distribution of utterences in a language or corpus. Often these utterences are broken down into single token sequences, and the probabilities are formed as a product of conditional probabilities.

$$\begin{aligned} p(y_1, y_2, ..., y_n) &= p(y_n|y_1, y_2, ...y_{n-1})p(y_1, y_2, ...y_{n-1}) & (2.2) \\ &= p(y_n|y_1, y_2, ...y_{n-1})p(y_{n-1}|y_1, y_2, ...y_{n-2})...p(y_1) \\ &= \prod_{i=1}^{n} p(y_i|y_1^{i-1}) & (2.3) \end{aligned}$$

where $y_1^{i-1} = \{y_1, y_2, ...y_{i-1}\}$ is the sequence of tokens from 1 to $i - 1$.

These probabilities can be calculated in different ways, or under different assumptions. An n-gram model assumes a Markovian conditional independence structure, where the next token is conditionally independent of all others, given the previous n:

$$y_t \perp\!\!\!\perp y_1^{t-n} \mid y_{t-n+1}^{t-1} \tag{2.4}$$

$$\therefore p(y_t|y_1, y_2, ..., y_{t-1}) = p(y_t|y_{t-n}, y_{t-n+1}, ..., y_{t-1}) \tag{2.5}$$

These models then estimate maximum likelihood probabilities from counts of occurences in corpora, and sometimes 'smooth' these probabilities to overcome the sparsity of their training data (Chen and Goodman, 1996).

Other models, such as neural language models, do not make such conditional independence assumptions and use a neural network as a function approximator:

$$p(y_1, y_2, ..., y_n) = g(y_1, y_2, ..., y_n) \tag{2.6}$$

In our work, we seek a *conditional* language model, which we can sample natural language given our conditioning code features $\mathcal{C}$. This conditional modelling is central to many tasks in natural language processing, which we elaborate in the next section.

2.1.2 Conditional Generation of Language

Many tasks in NLP require the generation of natural language conditioned on observations. Three of the largest are machine translation, automatic summarization, and captioning. In this section we briefly compare our task to these traditional fields, noting the similarities and differences with our task across each.

Machine Translation In machine translation, the objective is to generate the most likely sequence of a target language, $\{y_1, y_2, ..., y_m\}$, given a sequence in an origin language $\{x_1, x_2, ..., x_n\}$. The translation seeks to maximise the probability:

$$p(y_1, y_2, ..., y_m | x_1, x_2, ..., x_n)$$

Traditionally in machine translation, $\{x_1, ..., x_n\}$ and $\{y_1, ..., y_m\}$ correspond to different languages with the same semantic meaning. In this respect our task is not a typical translation one. However, since our models will at times attempt to generate one sequence from another, many techniques pioneered in this field are highly transferrable to our task. For further introductions to Machine Translation, we point the reader to Arnold (1994); Lopez (2008).

Summarization The task of summarization aims to extract summaries or descriptions from a document. A helpful review of the field is presented by Allahyari et al. (2017).

Typically this involves extraction and/or abstraction of a document in the same modality or language. Our task, generating descriptions about an element of code, naturally involves two different modalities - code as input, and english as output. However, in the best case our descriptions should reflect a true and relevant summary of the argument. Therefore, although ours is not a typical summarization task, it is sometimes framed that way in similar work (Iyer et al., 2016).

Captioning A final task with a parallel to ours is that of captioning. The distinction from summarization in this case is that captioning involves generating text from differnet modalities - often captioning image, (Vinyals et al., 2015b) or parts of one (Karpathy and Fei-Fei, 2015). The parallels with our code and language task are obvious here. However, it is perhaps unfair to describe documentation of code as 'caption' when it aims to be a description of only the most important parts of the code, and intentionally aims for succintness.

In summary our task falls in between a number of traditional NLP tasks and consequently related work blurs the boundaries of translation, summarization and captioning. In the next section we present an overview of the relevant theoretical components for our model, indicating their provenance from their respective fields.

2.2 Representation Learning

Given the maturity and prominence of neural networks , we assume a basic familiarity with the topic, including linear and sigmoidal layers, multilayer perceptrons, the backpropagation algorithm and stochastic gradient descent. For those seeking a further background on this topic, we strongly recommend a number of books Nielsen (2018); Goodfellow et al. (2016).

2.2.1 Distributed Representations

The vocabulary of natural language is vast and nuanced. Some words may appear very different lexically yet have similar semantic meaning - for instance 'big' and 'large'. Others may appear lexically similar, but have different meanings - 'large' and 'largesse'. In our language generation tasks we wish to generate words that have the correct semantics. As such it would be helpful to find numerical representations of words that reflect their semantic meaning.

One way of doing this is to distribute the semantic meaning over various components of a vector in a high dimensional space. In this space, 'big' and 'large' would be closer together than 'large' and 'largesse'. We could then use these pretrained representations, or *word vectors*, in our natural language modelling as a form of transfer learning (Hinton et al., 1986).

However, this begs the question: how do we etablish semantic equivalence in the first place? This is addressed the theory of distributional semantics which suggests that words which appear in similar contexts also have similar meanings (Harris, 1954). To quote linguist J.R Firth : *"You shall know a word by the company it keeps."* (Firth, 1957)

As a result, methods of training word vectors often revolve around word co-occurences. Traditional methods of doing this involved counting co-occurences of words in a corpus, and performing matrix decomposition on the co-occurence matrix (Deerwester et al., 1990). More recently neural networks have been used to train these vectors directly, using a local context window around the word in question, with much success (Mikolov et al., 2013a,b). An approach which combines both local window methods and matrix factorization methods is that of GloVe embeddings, by Pennington et al. (2014), which we use in our experiments. We make use of these embeddings to decode the words generated from our model, as their effectiveness has been demonstrated in a range of NLP tasks (Young et al., 2017).

2.2.2 Recurrent Neural Networks

A recurrent neural network (RNN) is a neural network with recurrent connections between neurons. These connections allow the network to pass on information as it processes a sequence (see Figure 2.1). In effect this forms a hidden state in the network, allows information from previous tokens to propagate inside the network and contextualise the processing of future tokens. Such networks have proved highly adaptable to many NLP tasks (Young et al., 2017), such as language modelling (T. Mikolov et al., 2011) and translation (Liu et al., 2014), due to the inherent sequential nature of the structure of language. They have also enjoyed widespread use within many other sequence learning settings (Lipton et al., 2015) .

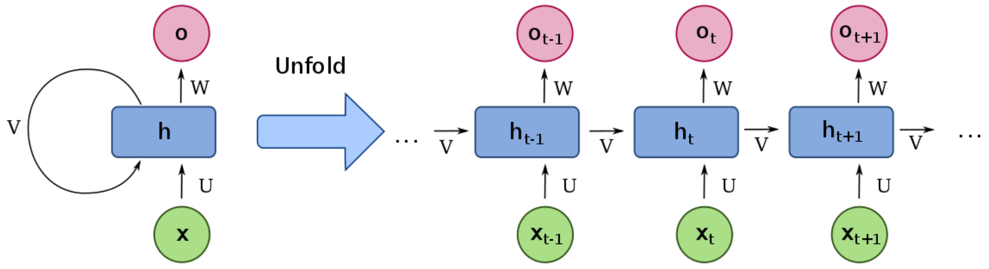


Figure 2.1: A diagram of an RNN.[1]This demonstrates how a recurrent connection can be 'unrolled' to form an effective hidden state maintained by the network as it processes as sequence.

An example of an RNN is the Long Short Term Memory Unit (LSTM) (Hochreiter and Schmidhuber, 1997). We use this RNN due to it's intrinsic ability to deal with the common training problems of vanishing and exploding gradients (Bengio et al., 1994). These comprise of four gating mechanisms, $(\mathbf{f_t}, \mathbf{i_t}, \tilde{\mathbf{c}}_\mathbf{t}, \mathbf{o_t})$, operating on the input $\mathbf{x_t}$ and state vectors $\mathbf{c_t}$ and $\mathbf{h_t}$. Each gating mechanism has a corresponding weight matrix, $\mathbf{W}$, and bias vector $\mathbf{b}$, and operate according to the following equations:

[1]Image sourced from https://en.wikipedia.org/wiki/Recurrent_neural_network

$$\mathbf{f_t} = \sigma(\mathbf{W}_f[\mathbf{h_{t-1}}, \mathbf{x_t}] + \mathbf{b_f}) \tag{2.7}$$

$$\mathbf{i_t} = \sigma(\mathbf{W}_i[\mathbf{h_{t-1}}, \mathbf{x_t}] + \mathbf{b_i}) \tag{2.8}$$

$$\mathbf{o_t} = \sigma(\mathbf{W}_o[\mathbf{h_{t-1}}, \mathbf{x_t}] + \mathbf{b_o}) \tag{2.9}$$

$$\tilde{\mathbf{c_t}} = \tanh(\mathbf{W}_c[\mathbf{h_{t-1}}, \mathbf{x_t}] + \mathbf{b_c}) \tag{2.10}$$

where [,] operator indicates concatenation and σ indicates element-wise sigmoidal function.

These gating equations then combine to update the cell state $\mathbf{c_{t-1}}$ and the hidden state $\mathbf{h_{t-1}}$ as follows:

$$\mathbf{c_t} = \mathbf{f_t} * \mathbf{c_{t-1}} + \mathbf{i_t} * \tilde{\mathbf{c_t}} \tag{2.11}$$

$$\mathbf{h_t} = \mathbf{o_t} * \tanh(\mathbf{c_t}) \tag{2.12}$$

where $*$ indicates element-wise multiplication, and tanh indicates element-wise hyperbolic tan.

Interpretting these operations gives a good insight into the success of the LSTM. Equation 2.7 represents the creation of a 'forget' gate, where vector $\mathbf{f_t}$ represents what fraction of each dimension of $\mathbf{c_{t-1}}$ to retain. Equation 2.8 creates an 'input' gate, where vector $\mathbf{i_t}$ represents what fraction of each dimension of our transformed input $\tilde{\mathbf{c_t}}$ we want to retain. $\tilde{\mathbf{c_t}}$ itself is created in equation 2.10. With both the 'input' and 'forget' gate, the the cell state is updated in 2.11. The new hidden state $\mathbf{h_t}$ then derives from this cell state, which is modified by the output gate $\mathbf{o_t}$. Both $\mathbf{c_t}$ and $\mathbf{h_t}$ pass forward to the next time step, while $\mathbf{h_t}$ is also out put by the cell. A diagram is presented in Figure 2.2.

LSTM's have shown a remarkable versatility and strength in NLP tasks (Young et al., 2017). In fact, a remarkable study of language modelling in 2017 showed that well-tuned LSTMs still surpass more recent and complex architectures, and achieve state of the art results, despite their relative simplicity (Melis et al., 2017).

[2]Image sourced from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

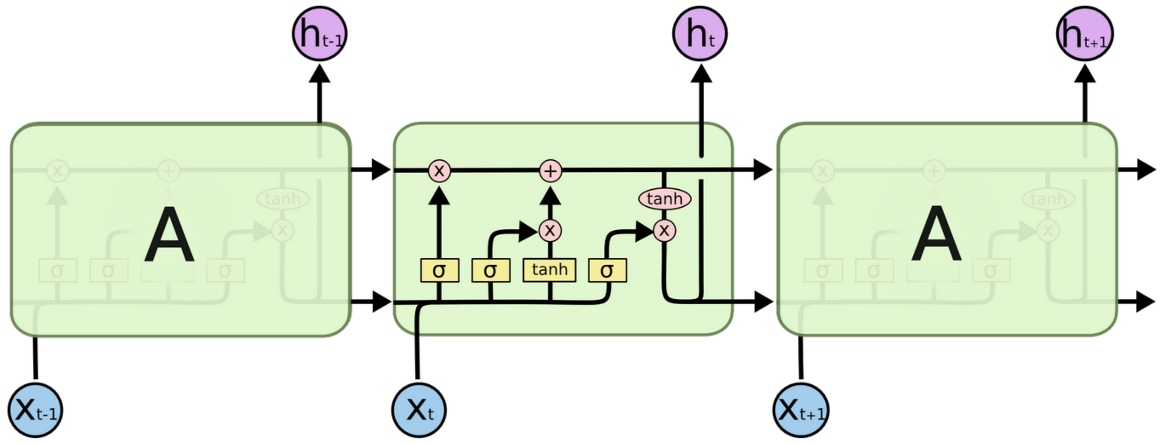


Figure 2.2: A diagram of an LSTM.[2]This demonstrates how the different gates operate to change the state of the LSTM. The top horizontal line indicates the $\mathbf{c_t}$ states and the bottom the $\mathbf{h_t}$ state. These operations are detailed in Equations 2.7 to 2.12.

There also exist other forms of RNN units. The Gated Recurrent Unit (Cho et al., 2014), is another recurrent network unit that aims to simplify the LSTM, and has shown comparable performance to the LSTM in tasks such as speech signal modelling and music modelling (Chung et al., 2014). However, for the purposes of our investigation, we use the LSTM for all recurrent units.

2.2.3 Sequence-to-Sequence Architectures

At each time step, an RNN unit accepts an input $\mathbf{x_t}$ and emits an output $\mathbf{h_t}$. In the translation context, this 1-to-1 mapping of input to output is the equivalent of translating a sentence before you hear the end of it. For languages such as German, where the verb comes at the end of the sentence, this presents a big challenge.

The sequence-to-sequence (Seq2Seq) architecture, originally proposed by Sutskever et al. (2014), overcomes this limitation by processing the whole input sequence before translation even begins. In this framework, presented in Figure 2.3, an RNN acts as encoder, through which all word tokens are passed. It acts as an encoder because its final hidden state is taken to be an encoding of the sequence into a intermediate representation.

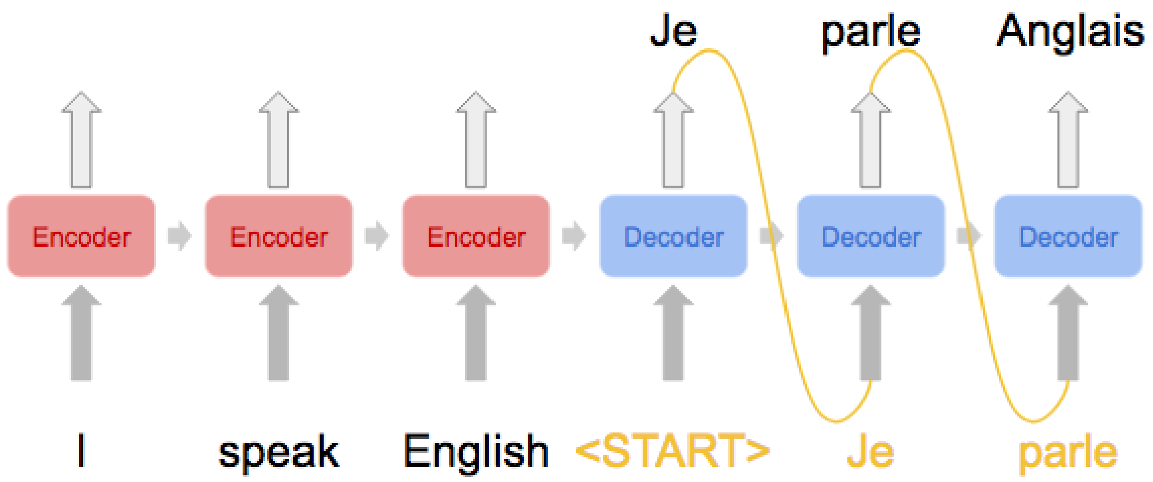


Figure 2.3: A sequence-to-sequence architecture. In this 'unrolled' example a single RNN acts as encodes words into an intermediate representation that is decoded by an RNN decoder. As inference time, the outputs of the decoder are fed back into the decoder, to generate the next word.

This representation is then used as the initial state of a decoder rnn cell, to conditionally generates a sequence of tokens.

First a "start-of-sentence" token is fed into the decoder, generating a distribution over tokens, conditioned on the initial state from the encoded vector. Then most likely token is chosen, and fed back into the decoder, generating a distribution over the next one. This method repeats until the sequence terminates, perhaps with a "end-of-sentence" token.

This describes a greedy form of decoding, where the most likely next word is continually selected from the decoder. It is also possible to explore the broader space of translations, by feeding in multiple tokens and retaining only the most likely sequences. This technique is known as beam search (Freitag and Al-Onaizan, 2017).

This architecture, though effective, has some drawbacks. It has been shown that as the source sequence gets longer, the performance of the model worsens significantly (Cho et al., 2014). This is partly due to the fact that model must now compress more information into the intermediate vector passed between encoder and decoder. A common way of dealing with such an issue is to use attention.

2.2.4 Attention

As the sequences get longer, encoder-decoder architectures struggle to pass the full information required from the encoder to the decoder. One way of improving this is to augment the decoding process with a contextual vector at each step. As before, the decoder estimates the conditional probability for the next word y_t, given the input sentence $\mathbf{x}$, and previous output tokens $y_1, ..., y_{t-1}$. However, this changes from being a function of simply the input token, y_{t-1}, and hidden state, s_t at step t:

$$p(y_t|y_1, ..., y_{t-1}, \mathbf{x}) = g(y_{t-1}, s_t) \tag{2.13}$$

to now being a function of the context vector at that point c_t aswell:

$$p(y_t|y_1, ..., y_{t-1}, \mathbf{x}) = g(y_{t-1}, s_t, c_t) \tag{2.14}$$

The context vector at each step is itself calculated using a weighted sum of the RNN outputs $\{h_1, ...h_{t_1}\}$:

$$c_t = \sum_{j=1}^{T_x} \alpha_{tj} h_j \tag{2.15}$$

where

$$\alpha_{tj} = \frac{\exp(a(s_{t-1}, h_j))}{\sum_k^{T_x} \exp(a(s_{t-1}, h_k))} \tag{2.16}$$

and a is a function that maps to the reals - often a multilayer perceptron that is trained simultaneous to main training.

This method was introduced by Bahdanau et al. (2014) who demonstrated that the networks could use the attention to align words during translation. In this case the α parameter acts like a weighting indicating the most important word used in each token generation - which word the decoder should pay attention to.

Variations on attention (as demonstrated in Figure 2.4) have found great use in the field of machine translation (Luong et al., 2015), and have even demonstrated their use independently of RNNs in this field (Vaswani et al., 2017). They have also have demon-

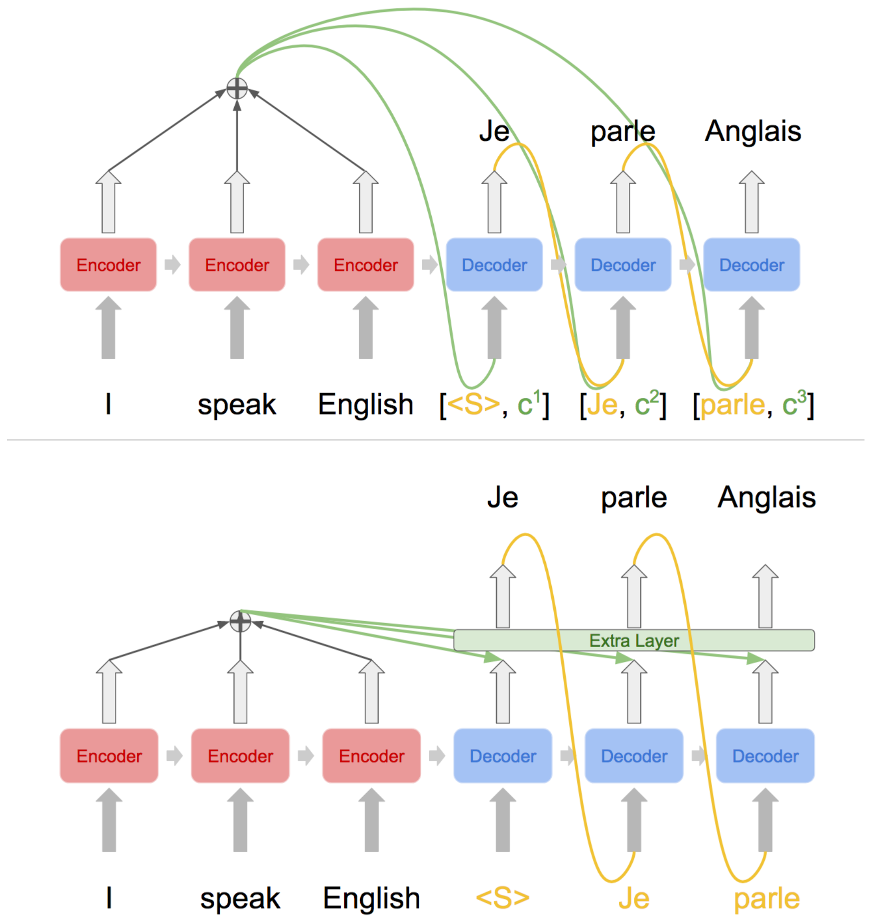


Figure 2.4: Two variations on attention in Seq2Seq architectures: Bahdanau et al. (2014) Attention (above), Luong et al. (2015) Attention (below). Both contextualise the decoding process, by building a vector from the weighted sum of encoder outputs. In Bahdanau attention, the vector is concatenated with inputs before RNN decoding. In Luong attention, the vector is concatenated with decoder outputs, before passing through another non-linear layer.

strated their effectiveness in fields such as captioning, where they prove an effectve way of working with different modalities (Xu et al., 2015).

In our models we use attention mechanisms extensively, both as a means of dealing with particularly long sequences (in our case of characters), and also as a means of dealing with the difficult modality of code.

2.3 Structure of Code

Although in principle the models presented in this thesis could be applied to any programming language, our dataset consists of functions in the Python prgramming language. In this section we provide a brief introduction to the language before presenting a broader overview of abstract syntax trees.

2.3.1 Introduction to Python

Python is a *dynamically-typed* language, meaning that the ‘type’ of an object, such as ‘string’ or ‘integer’, is be decided at runtime, rather than specified in advance, or *statically*. This allows a greater flexibility in the language, but comes at the expense of a less informative source code. Declaring a variable `var`, makes no guarantees on what it could be, or how it can be used.

Furthermore Python is *strongly-typed* language, meaning that a type can not be implicitly coerced into another one by the computer at run time. Changes in type must be explicit. For instance, evaluating `1 + '0'` would result in an `TypeError` being raised in Python. In *weakly-typed* JavaScript this returns `'10'`. This means that inappropriate use of variables can lead to run time crashes or errors very easily in Python.

Together, these features make documentation an especially important part of Python. Functions and classes therefore have a built-in convention for containing documentation, called a docstring. This is automatically displayed when the built-in `help` is called on the function. A typical function with a docstring is illustrated in Figure 1. Many open source libraries document their API's with HTML built straight from the docstring of the underlying methods.

There are no explicit rules for how docstrings must be written, but two popular conventions are those the Google[3] and numpy[4] conventions. These specify that the arguments of a function explicitly be described in the docstring, along with the description of the func-

[3]https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html#example-google
[4]https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html#example-numpy

```
def xkcd_palette(colors):
    """Make a palette with color names from the xkcd color survey.
    This is just a simple wrapper around the ``seaborn.xkcd_rgb`` dictionary.

    Parameters
    ----------
    colors :
        List of keys in the ``seaborn.xkcd_rgb`` dictionary.

    Returns
    -------
    palette :
        Returns the list of colors as RGB tuples in an object that behaves like
        other seaborn color palettes
    """
    palette = [xkcd_rgb[name] for name in colors]
    return color_palette(palette, len(palette))
```

Listing 1: A source code snippet from the validation set, with a numpy-style doctring. The docstring presented been trimmed for brevity. This format is widely used in scientific libraries.

tion. Our project involves sourcing and parsing functions that use these conventions, to be able to get the close mapping between the elements of the source code (the arguments) and their human written descriptions.

2.3.2 Abstract Syntax Trees

Programming languages communicate along two channels (Allamanis et al., 2017a). The *human readable channel* consists of the lexical names and words in files as read and written by humans. The *computer readable channel* consists of the information that forms the direct instruction set aimed at the computer. This direct instruction set is stripped of information superfluous to computer, such as comments and names, and is therefore difficult for humans to interpret directly. The instruction set is also represented by an intermediate structure, known as an abstract syntax tree (AST). This is directly obtained by parsing the written source code, and wholly specifies the instructional information to the computer. It is therefore the perfect data structure to investigate patterns in the computer readable channel.

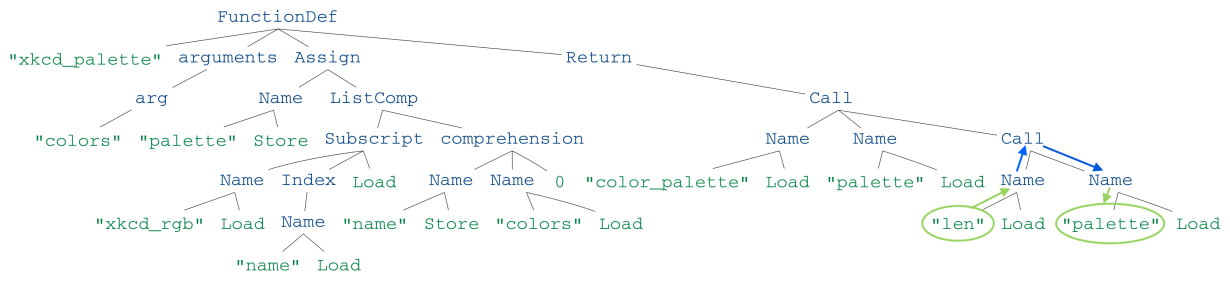


Figure 2.5: The Python AST corresponding to the code in Figure 1. We omit the node containing the docstring for clarity. Terminal nodes are shown in green, non-terminal nodes are shown in blue. We highlight a path from `"len"` to `"palette"`

$$(\texttt{"len"}, \text{string} \uparrow \text{Name} \uparrow \text{Call} \downarrow \text{Name} \downarrow \text{string} \,, \texttt{"palette"})$$

Figure 2.6: An example of a *path-context*, from the highlighted path in the above AST.

$$(\text{Name} \uparrow \text{Call} \downarrow \text{Name}, \texttt{"palette"})$$

Figure 2.7: Example 2.6 as a *variable path-context* for the variable `"len"`

The form of ASTs differ from language to language. Each language may have a different set of nodes and variables, and information associated with them. However, in general an algorithm that operates on one AST should be transferrable to another.

In our project, we work with the Python AST, as visible in Figure 2.5. This is composed of a series of instruction nodes, that terminate with lexical names. These names are the names of objects, which could therefore be methods, variables, classes, etc. In fact, in Python everything is an object.[5] As can be seen from the syntax tree, sometimes these object names are being loaded into memory and at other times other objects are being assigned to them.

In our project we aim to see if a statistical approach to looking at the structure or substructure of the AST can lead us to making inferences that will translate to language. For instance, the use of certain AST subtrees may suggest the type of an object or how its used in a function. These kind of inferences are powerful, because they should be robust to trivial lexical changes, like renaming a variable.

[5]https://jeffknupp.com/blog/2013/02/14/drastically-improve-your-python-understanding-pythons-execution-model/

2.3.3 Path-Contexts and Variable Path-Contexts

Although the AST is the definitive data structure that code is parsed into, it is also possible to reparameterise the tree as a set of shortest paths between terminal nodes. This is the reparameterisation that Alon et al. (2018c) developed to facilitate learning on syntax trees.

In this reparameterisation, an AST is made up of terminal nodes $\mathcal{T}$, and non-terminal nodes $\mathcal{N}$. A path p of length k, represented as $n_1 d_1 n_2 d_2 ... n_{k-1} d_{k-1} n_k$, where $n_i \in (\mathcal{N} \cup \mathcal{T})$, and $d \in \{\uparrow, \downarrow\}$. A *path-context* is then defined as a triple (x_s, p, x_f), where $x_s = val(n_1)$ and $x_s = val(n_k)$, $n_1, n_k \in \mathcal{T}$ and val returns the value of the node (Alon et al., 2018c) An example *path-context* from the furthest right of the example AST is shown in Figure 2.6.

The advantage of this representation is that it breaks down AST into a set of locally connected pieces, each involving two well-defined variables. This setup is highly desirable for our own investigations, except that we require the obfuscation one of the variables (our test argument). Therefore, for the purposes of this investigation we define two new context tuples: a *modified path-context*, and *variable path-context.*

A *modified path-context* is a triple (x_s, p', x_f) defined by *path-context* (x_s, p, x_f), where $p = n_1 d_1 p' d_{k-1} n_k$. In otherwords, the type of the leaf nodes are not included in the new path. This is done to remove redundant information, since the Python AST is so simple that the $val()$ of the node already specifies its type. The *variable path-context* for variable x_s is then defined as the double of (p', x_f) for the *modified path-context* from x_s to x_f.

Although these collections of *variable path-contexts* no longer specify the entirety of the tree, the ability to pluck out features of the code from the AST that only involve our particular argument will prove very useful in our investigations.

Chapter 3

Related Work

It is only through the recent development of large open source datasets and the techniques to process them, that statistical approaches to code-related tasks have become available to machine learning researchers. As would be expected of any new and growing field, the range of attempted tasks is still expanding.

As of writing no formal attempt has been made to automatically generate natural language descriptions of individual elements of code from its source. This is not surprising given how recently the field has developed, and the lack of a suitable dataset. However, progress has been made in the related fields of source code summarization, variable naming, documentation generation, and code language modelling (Allamanis et al., 2017a). These advances highlight possible approaches to statistically modelling structure of code like language.

In this review we summarise the current advances of these methods and how they relate to the task at hand. We start by examining progress in the language modelling of code, then examine sequence generation tasks, such as code comment prediction, code summarization or function naming. We then examine advances in the fields of representation of programs and abstract syntax trees. Finally we present an overview the existing datasets and the scope of the problems they are suitable to address, finding them lacking in our particular domain.

3.1 Language Models of Source Code

The earliest work modelling source code with natural language techniques comes from Hindle et al. (2012), who used simple Kneser-Ney smoothed (R. Kneser and H. Ney, 1995) n-gram models of code tokens, to create language models for large-scale Java and C projects. With these models, they were able to demonstrate that the cross-entropy of source code within projects was lower than that of large English corpora - indicating the presence of repetetive common patterns that could be leveraged for code completion, naming and summarization. This was consistent with findings by Gabel and Su (2010) who examined the lines of approximately 6000 projects of code and found widespread repetition of sections of up to several lines, both within and across projects. Despite the simplistic Markov chain assumption implicit in the ngram model, the effectiveness of this modelling techniques, especially within projects, opened up the field of code analysis to the wider natural language processing community.

Hindle's model, which only took into the lexical structure of code, was improved by Nguyen et al. (2013), by integrating semantic information into the n-gram model. Instead of training on the raw string of the token, the *lexeme*, this model condensed information such as data type, scope, role (such as literal, variable, function call) into a *sememe*, and trained an n-gram topic model, modelling both local context *sememes* ngrams, and global trends in the code. This highlighted the value of taking into account the semantic information in code, as well as the lexical, in prediction tasks.

Since then a number of different language models of code have been developed, largely finding their use in code-completion tasks. These have demonstrated the importance of factoring in the long range dependencies of code, and elements of the code beyond simple lexical structure. For instance Tu et al. added a cache mechanism to improve Hindle's ngram model in capturing longer range dependencies, while this itself was surpassed (up to 9 grams) with a recurrent neural language model by White et al. (2015). Most recently, Bhoopchand et al. (2016) used a sparse pointer network to create a language model that significantly outperformed a LSTM baseline on code completion tasks, that was able to refer to objects in code over 60 tokens previous.

This work in language modelling has direct applicability to our task at hand, as it points out relevant strategies in picking out the statistically important features of ‘natural’ code. In particular we note the importance of capturing long range dependencies (as seen in neural models), with the performance benefit that can be brought by taking into account semantic information (from the instructions given to the computer).

3.2 Sequence Generation from Source Code

Code summarization is a task that involves taking large sections of code blocks, and summarising their meaning in natural language. It has historically had parallels with the long running (inverse) problem of semantic parsing (Allamanis et al., 2017a; Zhong et al., 2017). Early approaches to this problem completely ignored the ‘naturalness’ properties of code and its comments, and instead was tackled with rule-based methods, static analysis, and human-crafted templates. (Sridhara et al., 2010)

However, the advent of the statistical approaches to code language modelling led by Hindle et al. (2012) signalled the start of similar approaches to sequence generation. The earliest is Movshovitz-Attias and Cohen, who used both ngrams and topic models such as LDA (Blei et al., 2003) to predict comments from JAVA source code. In this they found that modelling the lexical components of source code as coming from a mixture of topics - ‘code’ and ‘text’ - outperformed models that ignored this distinction. This pointed to the strength of taking into account features of code (such as distinguishing comments from commands), even if such destinction were still purely on the lexical level.

Since then neural models have become popular methods of attacking sequence generation, as they capture long range dependencies in sequences, and have shown great promise in other neural translation tasks.

A particularly successful example is that of Iyer et al. (2016), who applied a neural attentional model to the problem of generating code ‘summaries’. This model trained on

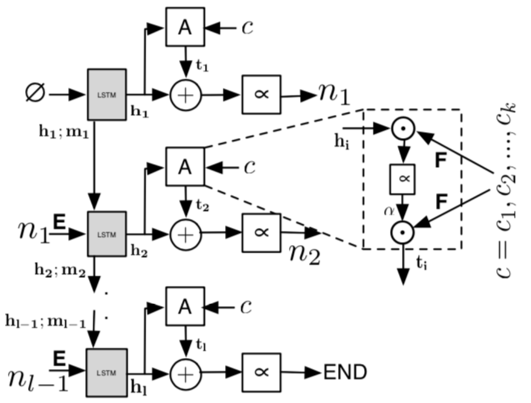


Figure 3.1: A diagram of CODE-NN, taken from Iyer et al. (2016). The A represents the attention mechanism over the sequence of code tokens $c_1, ..., c_k$, and $\mathbf{h_t}$ represents the hidden state of the LSTM. These are combined, and passed through a softmax to generate the next word token n_t. This is subsequently encoded by the LSTM to generate $\mathbf{h_{t+1}}$

a large collection of snippets and questions from Stack Overflow[1] , a programming help website. Iyer's model combined two features: a distributed representation of the code, generated by an attentional mechanism (Luong et al., 2015) over code token embeddings; and an LSTM unit (Hochreiter and Schmidhuber, 1997) to encode natural language tokens. Together these generated descriptions as a sequence of conditional distributions, in an encoder-decoder model. A visual schematic of the model is preseted in Figure 3.1.

Iyer et al then ran a beam search over the decoder to explore the space of likely sequences, and evaluated their generations using the BLEU-4 (Papineni et al., 2001) and METEOR (Denkowski and Lavie, 2014) metrics.

Iyer's model achieved a new record in the performance of the code summarization task. It outperformed rival NLP models such as MOSES (Koehn et al., 2007), a phrase-based translation model, and SUM-NN (Rush et al., 2015), another attention based summarization model using dense layers instead of LSTMs. It also achieved a first in learning to generate original sentences from arbitrary code sections, and has proved successful in

[1]https://stackoverflow.com/

other domains.

Loyola et al. (2017), for instance, adapted a similar attention model to Iyers to generate short descriptions of differnces in code. This time intead of training on a single piece of source code and questions, data was sourced to present pairs of code changes ('diffs') with comments describing the change ('commit messages'). In this setting, the attentional model was able to generate faesible messages, both within projects and between projects.

These attention models show the effectiveness of being able to pickout relevant portions of code at different points in sequence generation, but fail to take into account the longer term correlations of the tokens in source code. In fact by only processing source code with this kind of attention, both models treat the source code as little more than bag of token embeddings.

In this regard, an improvement on these attentional models is that of Allamanis et al. (2016), which uses a convolutional attention network over tokens, for the task of predicting the names of functions given their body. This is cast as 'extreme summarization'.

In this model, convolutions of fixed-width windows are run over the sequence of embedded source code tokens, creating a matrix of attentional feature vectors L_{feat}. These feature vectors are then element-wise multiplied with the hidden state of an rnn, $\mathbf{h_{t-1}}$, to be effectively 'selected' for their relevance to next generated token.

They are then normalised and convolved with an attention kernel, to give a set of attention weights, each corresponding to a token embedding. A final attention vector α is constructed from the convex combination of the token embeddings under the attention weights, and this vector is used to generate a distribution over the next function subtoken. An illustration of the algorithm is presented in Figure 3.2.

The authors of this model apply it to both the narrow task of generating function names, and that of code retrieval. In it they find that taking into account the long range features of the code in the convolutional model surpasses a baseline attentional model of the machine translation model originated by Bahdanau et al. (2014), and found that this performance was further improved by adding a caching mechanism to the model.

Both these attention models show the strengths that attention can provide in combining

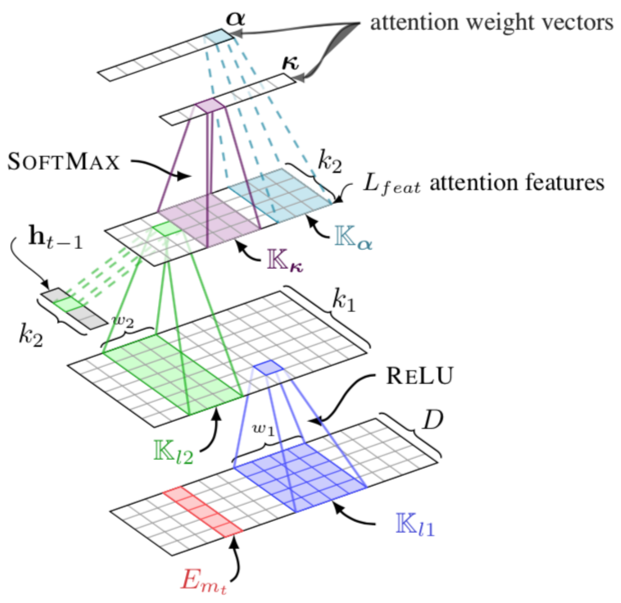


Figure 3.2: A diagream of the Allamanis et als convolutional attention mechanism taken from Allamanis et al. (2016). E_{m_t} represents an embedded token, which passes through a convolution layers with widow sizes w_1, w_2. It is then element-wise multiplied with $\mathbf{h_{t-1}}$ to give L_{feat} feature vectors, of dimension k_2. These are convolved to give attention vectors α and κ. (κ is only used in Allamanis' attention + caching model.)

the modalities of text and code. However, a key failing of both models remains their lack of appreciation for the syntax and semantics of the underlying code. Although Allamanis et al's model is more aware of the structure of lexical code tokens than Iyer et als, both models continue to ignore a fundamental channel of information communication through code - that of communication from human to computer.

As of writing no published work has been applied to generating documentation of source code using the syntactical structure of source code. However, a number of pieces of work have recently started to examine such structure in other tasks. These are the topics of the next section of review.

3.3 Syntactic Representations of Source Code

A number of probablistic models have attempted to capture the the syntax of code, outside of the field of documentation generation. Raychev et al. use decision trees to develop the probabilistic model of code from the AST, in a code-completion setting. Maddison and Tarlow (2014) use probabilistic context free grammars to model the AST, for a generative model. Neither of these approaches provides us with the natural structure we can easily use for sequence generation.

Allamanis et al. (2017b) models the AST as explicitly as a graph, adding extra edges such as “guarded by”, or “last lexical use”, to capture additional semantic information. These graphs are then processed with a Gated Graph Neural Network (Li et al., 2016) in a variable naming and variable misuse classification task. This last method proves remarkably effective, though it appears highly complex to implement, the additional semantic annotations added can appear particularly arbitrary. As a result of both of these, we avoid using these representations as inputs for our task.

An obvious way to model the AST is to try to capture the tree structure directly. This has mainly been applied in a program synthesis context, by adapting LSTM decoders to generate trees hierarchically (Dong and Lapata, 2016; Yin and Neubig, 2017). Recently, Chen et al. (2018) used a TreeLSTM (Tai et al., 2015) as in encoder and decoder to translate one programming language to another. However, thus far tree representations have not been used as encoders in a code-to-text context. Although this is a promising area of future work, in our task we prefer a more easily-decomposable representation of the AST to capture just the features related to our arguments.

This is provided by the Alon et al. (2018c)’s tree reparameterisation, introduced in the Background - the *path-context*. This paramaterisation has since been applied successfully in a function name classification context, in the Code2Vec model by (Alon et al., 2018b). In this model, each component of the *path-context* is embedded invidually, concatenated, and fed through a neural-network layer to creat into a path-context vector. A general code vector is then produced from a simple attention mechanism over the every path-context vector in the tree. With this representation, Alon et al are able to demonstrate a performance in a function name classification task that surpasses both Allamanis et al. (2016) and Iyer et al. (2016). This simplicity and proven effectiveness demonstrate an appealing approach to capturing semantics within the structure of with AST.

3.4 Existing Datasets

The scope of the tasks available to challenge researchers is naturally limited by the set of available datasets. In this section we outline the current state of the available datasets for researchers looking to generate sequences from source code. In doing so we indicate why none so far is suitable our task of generating documentation or summaries from fine-grained elements of source code.

3.4.1 Stack Overflow Dataset

A commonly used dataset for code captioning and summarization tasks is sourced from a large corpus of questions and code snippets from the programming help website Stack Overflow, as generated by Iyer et al. (2016). In it, questions such as "how do I concatenate entire result sets in mysql? ' are paired with the snippets of C# or SQL which are responses to the question, posted by other users online. Although widely used, this dataset suffers from a number for undesirable features for our task.

First of all the preparation of such a dataset demonstrates a lot of arbitrary cleaning. Since many of the responses from the website are invalid or do not quote code, the snippets in question have been found by searching for html `<code>` tags in the upvoted answers. The authors note that these sections of code often bore no real relevance to the question at hand. Therefore the authors were forced to train a semi-supervised classifier to filter only relevant snippets to the question at hand. This convoluted pipeline runs the risk of increasing the number anomalous datapoints in the dataset, whilst reducing almost a million *(query,snippet)* pairs from each language, down to 66,000 pairs of C# and 32,000 pairs of SQL.

Furthermore, the authors noted that often the informal code snippets often contained syntactic errors, with only 12% of SQL snippets parsing without syntactic error. They progress with a best-effort parse, but this lack of code quality naturally poses a problem for our fine grained analysis of code segments, or indeed any analysis using the syntactical properties of code.

Finally the dataset itself lacks a lot of the context and information needed for the granular analysis necessary for individual sections of code. Not only is the natural language not tailored to specific parts of the code, but the artificial snippets lack a lot of the context of real 'natural' codebases. In fact, given the snippets' only relevant context is that of the question, the authors are forced to mask items like string literals and the like, to prevent the close context of the question and nothing else. This modification of the code is also undesirable.

Naturally in our search for an appropriate code base, we seek larger bigger elements of 'real' parsable code, with appropriate descriptions of elements, and a greater sense of 'naturalness'.

3.4.2 Edinburgh Corpus

A recent corpus specifically tailored to code-to-text and text-to-code generation, is one of Python code released by Barone and Sennrich (2017), which we refer to as the Edinburgh Corpus. This is a set of 109,000 triplets of function declarations, bodies, and docstrings, scraped from the most popular projects in GitHub, using the same methodology as Bhoopchand et al. (2016).

In this dataset, the authors focus on finding code-in-the-wild, and extracting its docstring. Although these source code sections and natural language strings are more 'natural' than their Stack Overflow counterparts, they still don't suit our needs, mainly in the form of the language they provide.

The Python language enforces no constraints on the content or format of a class or function docstring. As a result, unless they follow the conventions laid out by numpy or Google, authors are not required to document arguments with a specific description, or even at all. Instead the docstring will likely just present an overall view of the function, with very little reference to the inner workings or details of the code. Without a section describing either arguments or specific elements in the code with their own natural language, it is impossible for us to attempt our close comparison task.

The major disadvantage of the Edinburgh corpus is that it does not contain enough of such sections, and where they do, they are difficult to extract from the rest of the docstring as a whole. As such this makes the Edinburgh corpus limited to our needs.

3.4.3 Other Datasets

A range of other datasets also exist in the code modelling field, often with specific tasks in mind. These are often utterly unsuitable from a natural language perspective.

For instance, a prominent corpus of data is the GitHub Java Corpus, by Allamanis and Sutton (2013). This corpus is composed of 14,800 open source Java projects, with over 350 million lines of code (LOC), and is approximately two orders of magnitude larger than the original dataset of Hindle et al. This dataset is rich and suited to its task of language modelling, but is of little use for natural language problems, given how poorly documented much open source code is.

Similarly Bhoopchand et al. (2016) dataset focuses on high quality code, by pulling large sections of Python from projects with more than 100 stars from GitHub, collecting 40 million LOC. Despite the quality and scale of this corpus, again the lack of appropriate documentation surrounding the code makes it unsuitable to our generation tasks, let alone anything as fine-grained as we would wish.

As a final example, Y. Oda et al. (9) source a particularly interesting dataset of individual lines of Python code from the Django open source project, with a corresponding line of human-written pseudocode. Although such datasets present interesting opportunities to look at close relations between code and language, the pseudo-code is far too close to actual code to help us in this task, and is highly artificial.

Chapter 4

The Dataset

In this section we present our novel dataset. We start with our motivation in collecting it, before moving onto the method of its collection and its fundamental structure. We then present a statistical analysis of the data collected, looking at the libraries it is composed of, its arguments and the accompanying code. We conclude by detailing the final preparations for its use in our experiments and present a summary of the datasets in Table 4.8.

4.1 Motivation

We have already motivated in Section 1.1 our desire to automatically generate documentation for individual elements of code: this would find great use as an IDE plugin for developers, whilst serving as a stepping stone for related tasks in code comprehension and type inference. However, in order to examine the automatic documentation of individual elements of code, we require an appropriate dataset that, as demonstrated in Section 3.4, does not currently exist.

Therefore we set out to collect our own dataset, according to a number of criteria:

1. The data should be 'real-world', ideally from open source code
2. The data should have a close alignment between natural language and the code described. Ideally the natural language *explicitly* describes the source code.

3. The code used must be high quality and parsable, so that syntactic structure can be extracted.
4. The data should require *minimal* preprocessing and cleaning, to avoid mistakes and errors.
5. The data should comparable in size to existing datasets

These goals were met in our collection of 40,000 function arguments with their respective natural language description, from 112 of the most popular libaries in 'PyPI'[1], the Python Package Index - where open source python libraries are deployed. In the rest of this section we outline how we obtained this dataset, and the structure of the data within it. As this is a new and previously unseen dataset, we also present an analysis of its composition, both qualitatively and quantitatively. Finally we present the different partitions of the dataset, we used in our experiments.

4.2 Method of Collection

As mentioned in Section 2.3.1, a number of conventions exist for the formatting of docstrings, the two most prominent ones being numpy-style and Google-style. Not all codebases use such styles, (as seen in the Edinburgh Corpus (Barone and Sennrich, 2017)), but they are popular with large open source projects.

Sphinx[2] is the industry standard for generating static HTML pages of documentation from source code. It contains a plugin - *napoleon*[3] - especially designed for Python docstrings formatted according to the numpy or Google conventions. This plugin loads the source into memory and obtains the docstring from the AST itself. Then *napoleon* uses its custom parser to parse out different features from the docstring, such as argument names and their descriptions.

In order to make use of *napoleon* docstring parsing facilities, we wrote a plugin, *bona-*

[1]https://pypi.org/
[2]http://www.sphinx-doc.org/
[3]https://sphinxcontrib-napoleon.readthedocs.io/en/latest/

parte[4], forking the *napoleon* source. Instead of generating HTML, this plugin generated a series of yaml[5] files with the associated data we required, including argument names, descriptions, docstrings, function code and filenames.

```
-   argument_name: 'tensors'
    argument_description: ' a list of variable or op tensors.'
    function_name: 'add_zero_fraction_summaries'
    function_args: ['tensors', 'prefix']
    function_signature: '(tensors, prefix=None)'
    library: 'tensorflow'
    filename: '/tensorflow/contrib/slim/python/slim/summaries.py'
    other_argument_info:
        'prefix': {desc: ' An optional prefix for the summary names.' }
    docstring: |
      Adds a scalar zero-fraction summary for each of the given tensors.

      :param tensors: a list of variable or op tensors.
      :param prefix: An optional prefix for the summary names.

      :returns: A list of scalar `Tensors` of type `string` whose contents are the
                serialized `Summary` protocol buffer.
    src: |
      def add_zero_fraction_summaries(tensors, prefix=None):
        """<docstring>"""
        summary_ops = []
        for tensor in tensors:
          summary_ops.append(add_zero_fraction_summary(tensor, prefix=prefix))
        return summary_ops
```

Listing 2: An illustrative example of a single data point. The docstring in the source has been elided for brevity and replaced with the <docstring> tag. A full table of the field names and types is presented in Appendix Table A.1

Having written the plugin, we required a repository of large scale projects that documented their code according to these conventions. Therefore, we ran the bonaparte plugin on the 300 most popular libraries in PyPI, as of April 2018 [6]. We then processed our data by removing arguments without descriptions (such as `self`) or without code bodies (from abstract base classes), or with an inappropriate documentation convention. This fulfilled our objective of obtaining a large number of arguments, descriptions and source code

[4]https://github.com/condnsdmatters/bonaparte
[5]http://yaml.org/
[6]https://python3wos.appspot.com/

from popular high quality code-in-the-wild sources, with minimal processing to extract the relevant data.

4.3 Structure of Data

The dataset comprises of a list of Python function arguments, their natural language descriptions and surrounding metadata. This metadata is extensive, containing numerous fields including the source code of the function, function name and filename, among others. An example datapoint is presented in Listing 2. The full structure of the data fields is also present in the appendix, Table A.1.

4.4 Analysis of Data

4.4.1 Statistical Analysis of Libraries

We first analysed our dataset by looking at the libraries included within it. We found that although the criteria for choice of library was based solely on frequency of downloads from PyPI, there was a clear bias towards scientific libraries in the composition of the data.

Of the 112 libraries that eventually contributed data to the dataset, only 21 are labelled with the 'Scientific/Engineering' tag on PyPI, yet these libraries contribute 68.3% of arguments to the overall dataset, and 61.6% of the functions. Even more surprisingly one library, `tensorflow`, contributes 41% of the arguments to the data - vastly more than any other library. In fact, as can be seen from Table 4.1, it contributes almost six times as many arguments as the second placed library - coincidentally also developed by Google.

This outsize contribution from relatively few libraries is largely the result of the design of these libraries and their use cases. Scientific libraries have vast APIs, and users of the libraries often require different interfaces for potentially similar functions. As an example, two library methods such as `conv2D` and `conv3D` may in an abstract sense perform very similar functions - a mathematical convolution - yet are critically distinct in a scientific

Library	Arguments	Functions	Scientific
tensorflow	**17002**	**4340**	**True**
google	2829	1098	False
scipy	2030	549	True
networkx	1869	669	True
matplotlib	1457	366	True
sklearn	1403	368	True
pandas	1317	419	True
magenta	1045	357	False
($100 < arguments < 1000$)	10172	3137	True: 6, False: 26
($0 < arguments < 100$)	2300	1036	True: 9, False: 84
TOTAL	41424	12339	True: 21, False: 112

Table 4.1: Break down of the largest libraries in the dataset, by contribution of arguments. Despite their relative infrequency in the overall dataset, scientific libraries contribute the majority of arguments (68.3%) and functions (61.6%). Of all the libraries `tensorflow` contributes by far the most data - 41% of the arguments.

setting. The necessity of these kinds of distinctions, and the precision of their use case, leads these libraries to have large numbers of exposed functions, requiring clear documentation.

	Top Argument Names	Count	Top Function Names	Count
1.	**name**	**1917**	fit	122
2.	x	439	transform	102
3.	kwargs	303	train	77
4.	axis	263	evaluate	71
5.	dtype	260	get	61
6.	a	230	conv2d	59
7.	G	227	Client	52
8.	inputs	224	update	46
9.	input	223	add	45
10.	value	210	specgram	45

Table 4.2: The most popular variable names and function names in the raw dataset. As can be seen these are either dominated by scientific terms and single letter variable names, such as those in scientific libraries (x, axis, dtype). Scientific functions which take large numbers of arguments contribute heavily to the dataset. The variable "name" provides an outsize contribution, due its repeated presence in `tensorflow`, often with the same description.

4.4.2 Statistical Analysis of Arguments

Corresponding to our findings on a library level, we found that a large number of function and argument names were very scientific in nature. Table 4.2 illustrates this with a list of the most popular argument names and argument function names in the dataset.

Furthermore we found a lot of repetition of the same argument name within the dataset, as can be seen from the histogram in Table 4.3. Reassuringly we found that, aside from having different source code, these duplicates would also often have different descriptions and came from different packages. A histogram of counts of unique (name, description) pairs verifies this in Table 4.4, while some illustrative examples of the different descriptions and packages for the most popular names are presented in Appendix Tables A.2 and A.3.

We notice at this point that our distributions are heavily outweighed by one repeated (name, description), which accounts for 3.4% of the whole data. This argument is from tensorflow: `name` - *"a name for the operation (optional)"*. This outlier is visible too in our histogram of description lengths, (Figure 4.1), which otherwise shows a reasonable distribution of short descriptions. We note this outlier to inform our final preparations.

Apart from this we note no major irregularities the arguments. The distribution of length of names is presented in Figure 4.1, showing most are under 10 characters long. We also note that the average number of arguments per function is a 3.836 ± 2.96, indicating that most most functions supply a modest number of arguments to the dataset.

4.4.3 Statistical Analysis of Code

Our dataset of derives from 264,056 lines of code (LOC)[7]. When each function is split into different arguments, the total LOC associated with the dataset grows to 1,340,891. The distribution of these lengths is visible in figure 4.2, demonstrating that most functions are shorter than 50 lines of code. The average length of the source code for a datapoint is 34 LOC, with a standard deviation of 57 LOC.

Since we aimed to investigate the code using Alon et al. (2018c)'s path based parame-

[7]Calculated with the `cloc` command line tool: https://github.com/AlDanial/cloc

Repetitions, N	Names Repeated N Times (% of Data)
1	11.74
2 - 4	17.30
5 - 9	10.99
10 - 19	11.56
20 - 99	26.12
100 - 499	16.27
500 - 1999	6.11

Repetitions, N	Names Repeated N Times (% of Unique Names)
1	53.55
2 - 4	31.15
5 - 9	7.84
10 - 19	3.96
20 - 99	3.08
100 - 499	0.41
500 - 1999	0.02

Table 4.3: **Left**: the % of data composed of arguments with names repeated N times
Right: the % of unique argument names that are repeated N times
These indicate that approximately half of the dataset is made up of names that occur over 10 times, and only 12% of arguments have a unique name (*left*). However these account for less than 8% of the unique names in the dataset (*right*). A single outlier - the variable name `name` from `tensorflow`, accounts for 6.1% of the collected data

Repetitions, N	(Name, Desc) Repeated N Times (% of Data)
1	47.691
2 - 4	28.881
5 - 9	9.483
10 - 99	10.574
100 - 499	0.000
500 - 1999	3.371

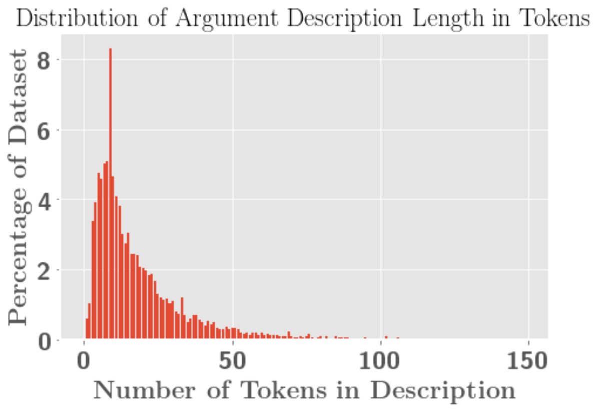
Repetitions, N	(Name, Desc) Repeated N Times (% of Unique (Name, Desc))
1	77.009
2 - 4	19.561
5 - 9	2.472
10 - 99	0.953
100 - 499	0.000
500 - 1999	0.005

Table 4.4: **Left**: the % of data composed of args with (name, desc.) pairs repeated N times
Right: the % of unique (name, desc.) pairs that are repeated N times
Almost half the datapoints have a unique name + description, indicating a variety of descriptions for the same name (*left*). However `tensorflow`'s `name`: *"A name for operation (optional)."* still accounts for 3.3% of the collected data.

Repetitions, N	CodePaths Repeated N Times (% of Paths)
1	2.195
2 - 9	12.056
10 - 99	22.349
100 - 999	26.367
1000 - 9999	30.418
10000 - 99999	6.615

Repetitions, N	CodePaths Repeated N Times (% of Unique CodePaths)
1	33.753
2 - 9	51.099
10 - 99	13.445
100 - 999	1.510
1000 - 9999	0.189
10000 - 99999	0.005

Table 4.5: **Left**: the % of all codepaths that are repeated N times
Right: the % of unique codepaths that are repeated N times
About 15% of unique codepaths are repeated more than 10 times (*right*) yet these account for $>$ 85% of all codepaths (*left*). This indicates a small number of paths are exceptionally common, yet the range of different paths is relative great.



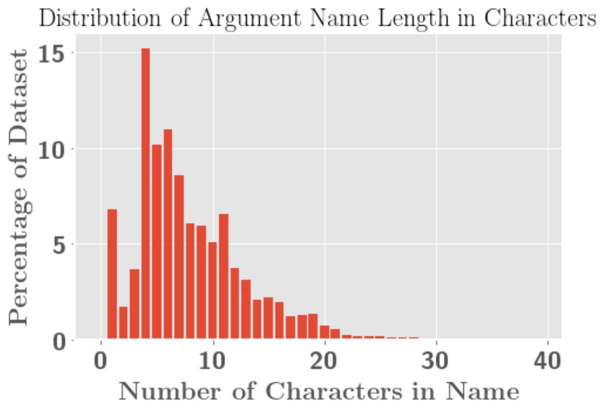


Figure 4.1: The frequency distributions of description lengths (*left*) and argument name lengths (*right*). `tensorflow`'s `name` outlier is visible in the descriptions graph.

terisation, we also investigated the most popular paths present in the dataset. Specfically, for each argument, we extracted the *modified path-contexts* for which the argument was a terminal node, and counted the paths associated with these contexts. A table histogram, Table 4.5, displays the distributions of paths. In it one can see that very few paths in the dataset are completely unique, which is promising if we are hoping to generalise across different sets of paths. Furthermore, looking at the number of paths per data point (Table 4.7), we see most data points have a large number of paths - about half have between 50 and 500 paths, which is also promising.

However, we also note from this table that some outliers have a preposterous large number of paths. These often correspond with the outliers that have exceptionally large line of code, and are often tied to items like class definitions. None-the-less, this indicates the data set is very much source from 'real-life'.

4.5 Final Preparations

Our final preparations of the dataset involved the filtering of duplicates that we had discovered in Section 4.4, and creating two different partitionings of train, validation and test according to 'in-project' splits and 'out-project' splits.

	Most Popular Paths	% of Data
1.	Name ↑ keyword ↑ Call ↓ keyword	0.679
2.	Name ↑ Call ↓ Name	0.672
3.	Name ↑ keyword ↑ Call ↓ keyword ↓ Name	0.647
4.	Name ↑ Call ↓ Attribute	0.296
5.	Name ↑ Call ↓ Attribute ↓ Name	0.241
6.	Name ↑ keyword ↑ Call ↑ Assign ↑ If ↓ Assign ↓ Name	0.229
7.	Name ↑ AugAssign ↑ For ↑ FunctionDef ↓ Assign ↓ List ↓ List ↓ Num	0.210
8.	Name ↑ keyword ↑ Call ↑ Assign ↑ If ↓ Try ↓ Assign ↓ Call ↓ Name	0.192
9.	Name ↑ Call ↑ Assign ↑ Try ↑ If ↓ Assign ↓ Name	0.189
10.	Name ↑ Call ↑ Assign ↑ Try ↑ If ↓ Assign ↓ Call ↓ keyword	0.184

Table 4.6: The most popular the paths of the *modified path-contexts* of the dataset

Paths Per DataPoint	Count	%-ile of data points
0	24	0.079
1	138	0.531
2 - 9	2451	8.563
10 - 49	8717	37.131
50 - 99	4517	51.934
100 - 499	10023	84.781
500 - 999	2449	92.807
1000 - 9999	2160	99.885
10000 - 99999	32	99.990
100000 - 999999	3	100.000

Table 4.7: A table of the number of codepaths per argument. This spread covers several orders of magnitude, although 50% of arguments fall between 50 and 500. Having 0 paths indicated an abstract class method which raised an `NotImplementedError`. These were filtered out.

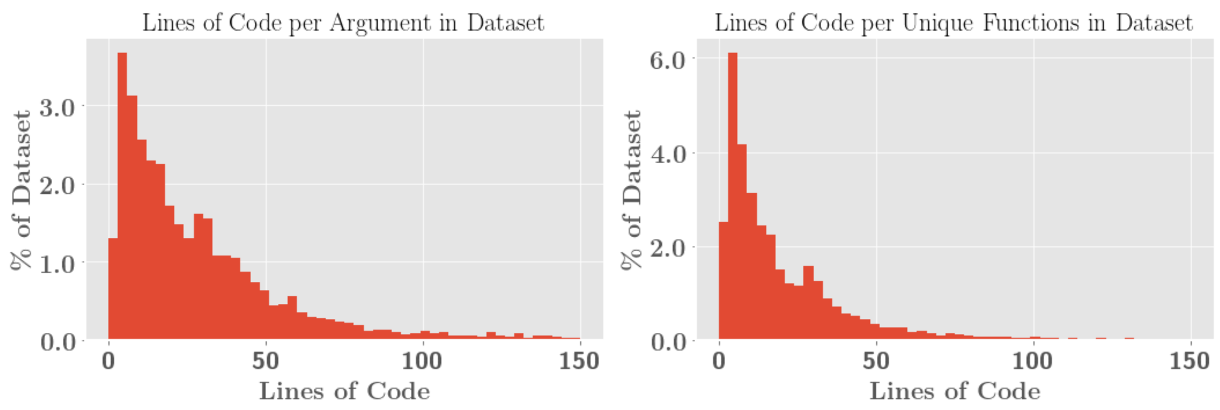


Figure 4.2: **Left**: distribution of lines of code in the function of each argument
Right: distribution of lines of code of the original functions in the dataset
These indicate that most functions in the dataset are under 25 LOC (*left*), yet there are still a significant number of arguments with functions over 75 LOC.

First we restricted the number of duplicates of (name, description) pairs to 10. This removed the problem of exemplified by tensorflow's `name`, while still maintaining a 91% of the data. This would be a **Full** dataset, which would be used to investigate AST related approaches, since the ASTs would be different for each of the duplicates. We also prepared a **Reduced** dataset, where (name, description) pairs where unique. This would give us the opportunity, to do a broader analysis, where we might not use the AST.

Then we partitioned our datasets into train, validation and test in two different ways:

1. **Random Split** - we partitioned the arguments randomly
2. **Library Split** - we partitioned according to library, and choose library randomly

In both cases we aimed to maintain a ratio of 7:1:2 for the train, validation and test. In the second case, the partition by library was not entirely random - due to its size, tensorflow was required to remain in training data, as too were some numpy and scipy, due to tensorflow's occasional use of their code. The rest of the libraries were chosen at random. This partition would allow us to investigate the difference of our algorithms in 'in-project' and 'out-of-project' situations.

Our final datasets are presented in Table 4.8, with their train and validation splits.

Name	Train	Validation	Test
Full Random Split	24886	3551	7114
Reduced Random Split	16042	2289	4586
Full Library Split	23969	3856	7726
Reduced Library Split	15216	2565	5138

Table 4.8: Our Final Datasets. Random Split partitions training, validation and test at random, while Library Split ensures test and validation are from different libraries to training. A Reduced dataset indicates (name, description) pairs are unique, and is therefore useful for investigations that dont use the AST.

Chapter 5

Method

In this section we first present a summary of our areas of investigation. We then present our models compositionally, first with an overview and then with a detailed description of each of the components. Finally we outline three sections that are key to the implementation of our project: the tokenization of the data, the evaluation method, and the general experimental procedure.

5.1 Areas of Investigation

In our investigations we focus on two main aspects of the code to generate descriptions for our arguments. These are the names in the function's signature, and the *variable path-contexts* (VPCs) in the function's AST.

In our experiments, we model the names of the signature as sequences of characters. We do this to capture the fact that variable and function names can often be composite and abbreviated - e.g. `web_ctx`. In this case both parts of the name might indicate a different clue to the argument: the `web` prefix may indicate a use in the internet domain; the `ctx` suffix may indicate a context object. In designing our models, we aim to pick up these patterns and conventions on the character-level.

We also focus on generating descriptions solely from the argument's VPCs. We feel

Model	Uses Signature Data	Uses VPC Data
Rote Learner	✓	✓
Seq2Seq Model	✓	
Code2Vec Decoder		✓
Code2Vec + Seq2Seq Model	✓	✓

Table 5.1: A Summary of the Models in our Investigation

that this is the most robust way of drawing inferences from the code, as it only examines instructions given directly to the computer. Any inferences here would be invariant under transformations such as renaming of variable. We hoped that by examining all the VPCs present for an argument, the model would learn representation for variables that indicate common usage (such as which methods are called on it), or perhaps even type.

We prepared four models to investigate these data: A Rote Learner to act as baseline for our investigations; a character-level Seq2Seq Model to investigate the signature names; an original Code2Vec Decoder model to investigate the VPCs; and a Code2Vec + Seq2Seq Model to investigate both inputs combined.

5.2 The Models

5.2.1 Rote Learner Model

Our Rote Learner model was designed to act as a strong benchmark in all our investigations. It was designed according to a simple principle: *the Rote Learner generates a description from a test point by returning in full a random description from a list of best-matching training point.* It is defined formally in Algorithm 1.

The benefit of this model is that the definition of 'best-matching' can then be changed according to the modality of the input data. In the sections below we present a brief description of these matching algorithms, with an appendix displaying each algorithm in pseudocode. A summary of these matching algorithms is presented in Table 5.2,

1. **NCharacter-Gram Overlap** *(Signature Data)*

Algorithm 1 The general Rote Learner algorithm

```
procedure GenerateDescription($t$)    ▷ Generate a description for test argument $t$
    $\mathcal{M} \leftarrow$ BestMatchingSet($t, training\_points$)
    $x \leftarrow$ RandomChooseOne($\mathcal{M}$)
    $d \leftarrow$ GetDescription($x$)
    return $d$
end procedure
```

This matching criterion was used for function signature data, and so operates on sequences of characters. It compiles a list of the training points which have the longest n character overlap with the test data point (or the longest common substring). If the same point has two of such overlaps, it is included twice, and so forth.

2. **Proportional Contexts** *(VPC Data)*

   This criterion matches VPCs by treating each VPC as an atomic unit. For each VPC within the test point, the criterion finds all the training points with the same VPC. It combines these into one list, returns this list.

3. **Max Contexts** *(VPC Data)*

   This criterion takes the list output by the **Proportional Context** criterion, and only chooses the training points that appear most frequently in the list.

4. **Proportional SubContexts** *(VPC Data)*

   This criterion also match VPCs of the data, but treats each VPC as sequence of nodes. This allows it to match subpaths. The matching criterion takes the test point, and for each VPC within it, finds the training VPC with the longest subsequences that matches it. It collects the corresponding datapoints to these paths and combines them into one list, and then returns this list.

5. **Max SubContexts** *(VPC Data)*

   The matching criterion takes the list output by the **Proportional SubContext**, and only chooses the training points that appear most frequently in the list.

6. [**Combinations**]*(Signature Data + VPC Data)*

Naturally these criteria can be combined across modalities to act upon combinations of the function signature and AST. In cases where this is done, the new list is simply the combination of the lists from the combined individual criteria.

BestMatchingSet	Use Sig.	Use AST	Description: *choose list of training points...*
NCharacter-Gram Overlap	✓		which have the longest n-character gram overlap with test
Proportional Context		✓	which match each VPC in test, and combine
Max Context		✓	which match each VPC in test, combine, and take most frequent
Proportional SubContext		✓	which *best-effort* match each VPC in test, and combine
Max SubContext		✓	which *best-effort* each VPC in test, combine, and take most frequent
Combinations	✓	✓	from the combined the lists from each BestMatchingSet

Table 5.2: A summary of the BestMatchingSet functions used in our RoteLearner (Algorithm 1). For sake of simplicity, a *best-effort* match here refers to the match with longest matching subsequence, when treating a VPC as one long sequence

5.2.2 Seq2Seq Model

Overview

Our character level sequence-to-sequence model follows the standard formulation as found in Sutskever et al. (2014) and Bahdanau et al. (2014), but encodes a sequences of characters, and decodes a sequence words. We use a bidirectional LSTM as our encoder, concatenating its outputs(Bahdanau et al., 2014), and use a single LSTM as a decoder. We also use a Luong attentional mechanism (Luong et al., 2015) over encoder outputs, throughout our decoding.

Since our vocabulary of characters is size 70, we embed our input into 70-dimensional space, with trainable embeddings. We initialising these with a one-hot encoding. Our characters are embedded into a 200 dimensional space use GloVe embeddings for our decoder.

These are not made trainable, due to concerns that the model may overfit. We apply dropout (Srivastava et al.) to inputs, outputs and states in both the encoder and decoder, and clip gradients to magnitude 1. The dropout fraction is a tuned hyperparameter. We define our object function as the minibatch average of the summed cross-entropy losses between the true words and the distributions over words at each time step. We train our model by minimising this with Adam (Kingma and Ba, 2014).

We have presented the ideas behind these models in the Background and present the model in full mathematical formality below. In it we present the encoder as uni-directional, and treat the modification for bidirectionality separately, since we experiment with both variations of model.

Encoder

First we iteratively embed each sequence of input tokens $\{x_1, ..., x_k\}$ into its vector representation $\{\mathbf{x}_1, ...\mathbf{x}_k\}$, according to:

$$\mathbf{x}_t^T = \mathbf{H}^C_{x_t,:} \tag{5.1}$$

where $\mathbf{H}^C \in \mathbb{R}^{V_c \text{x} D_c}$ is our character embedding matrix, V_c is our character vocabulary, D_c is our encoding dimension, and $\mathbf{X}_{i,:}$ indicates selection of row i from a matrix $\mathbf{X}$.

Then we define an LSTM with functions f and g, which takes input vector $\mathbf{x}_t$ and generates an output, $\mathbf{h}_t$, and updates its own state $\mathbf{c}_t$ according to Background Equations 2.7 to 2.12.

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}) \tag{5.2}$$

$$\mathbf{c}_t = g(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}) \tag{5.3}$$

With these sequences of vectors we then encode each vector $\{\mathbf{x}_1, ..., \mathbf{x}_k\}$ with the LSTM until we have generated outputs $\{\mathbf{h}_1, ..., \mathbf{h}_k\}$ and a intermediate fixed length representation,

s of the sequence:

$$s = (\mathbf{h}_k, \mathbf{c}_k) \tag{5.4}$$

Decoder

We then seek to iteratively decode the our sequence of word vectors $\{\mathbf{y}_1, ..., \mathbf{y}_l\}$ given this state s such that:

$$P(\mathbf{y}_1, ..., \mathbf{y}_l) = \prod_{i=1}^{l} p(\mathbf{y}_i | \{\mathbf{y}_1, ..., \mathbf{y}_{i-1}\}, s) = \prod_{i=1}^{l} d(\mathbf{y}_{i-1}, \mathbf{h}'_{i-1}, \mathbf{c}'_{i-1}) \tag{5.5}$$

where $\mathbf{h}'_i, \mathbf{c}'_i$ are the hidden states of the decoder LSTM at step i, $\mathbf{y}_0$ is start-of-sentence vector, and d is a general decoding method using the LSTM which we illustrate below.

First we initialise our decoders hidden state ($\mathbf{h}'_0$,$\mathbf{c}'_0$) with the our intermediate state s:

$$(\mathbf{h}'_0, \mathbf{c}'_0) \leftarrow s = (\mathbf{h}_k, \mathbf{c}_k) \tag{5.6}$$

Then we iteratively generate $\mathbf{y}_t = d(\mathbf{y}_{t-1}, \mathbf{h}'_{t-1}, \mathbf{c}'_{t-1})$. We do this by performing:

$$\mathbf{h}'_t = f(\mathbf{y}_t, \mathbf{h}'_{t-1}, \mathbf{c}'_{t-1}) \tag{5.7}$$

$$\mathbf{c}'_t = g(\mathbf{y}_t, \mathbf{h}'_{t-1}, \mathbf{c}'_{t-1}) \tag{5.8}$$

$$\mathbf{w}_{t+1} = softmax(\mathbf{W}_p \mathbf{h}'_t) \tag{5.9}$$

$$w_{t+1} = argmax(\mathbf{w}_{t+1}) \tag{5.10}$$

$$\mathbf{y}^T_{t+1} = \mathbf{H}^W_{w_{t+1}:} \tag{5.11}$$

where $\mathbf{H}^W \in \mathbb{R}^{V_w \text{x} D_w}$ is our word embedding matrix, $\mathbf{W}_p \in \mathbb{R}^{V_w \text{x} L}$ is a projection matrix, V_w is our word vocabulary size, D_w is our word embedding dimension, $\mathbf{h_i} \in \mathbb{R}^L$, L is the dimension of our decoding LSTM, and $\mathbf{X}_{i,:}$ indicates selection of row i from a matrix $\mathbf{X}$.

The above routine is performed at inference time. However, during training we choose

w_{t+1} to be the next word token in the true sequence $\{y_1, ...y_l\}$, instead of the most likely next word generated by the decoder. Therefore in this case $w_{t+1} = y_{t+1}$ **not** $w_{t+1} = argmax(\mathbf{w}_{t+1})$. This is known as *teacher forcing* and is known to produce better results for these models (Goodfellow et al., 2016; R. J. Williams and D. Zipser, 1989).

Bidirectional LSTM

The above framework is modified when dealing with a bidirectional LSTM encoder. The bidirectional LSTM consists of two LSTMs, one which processes the vectors forward $\{\mathbf{x}_1, ...\mathbf{x}_k\}$, producing $\{\overrightarrow{\mathbf{h}}_1, ...\overrightarrow{\mathbf{h}}_k\}$, and one which processes the vectors in reverse order $\{\mathbf{x}_k, ...\mathbf{x}_1\}$, producing $\{\overleftarrow{\mathbf{h}}_1, ...\overleftarrow{\mathbf{h}}_k\}$.

Following Bahdanau et al. (2014)'s example we concatenate these vectors to make a new vector of twice the length. Otherwise, the model is identical:

$$(\mathbf{h}_i^b)^T = [\overrightarrow{\mathbf{h}}_i^T, \overleftarrow{\mathbf{h}}_i^T] \tag{5.12}$$

$$(\mathbf{c}_i^b)^T = [\overrightarrow{\mathbf{c}}_i^T, \overleftarrow{\mathbf{c}}_i^T] \tag{5.13}$$

$$s = (\mathbf{h}_k^b, \mathbf{c}_k^b) \tag{5.14}$$

Attention

To make the adjustment of adding attention, we define a modification to the above decoding sequence. Since we use Luong et al. (2015)'s global attention, we incorporate a contextualising vector, $\mathbf{z}_t$, to the output of our LSTM at each step, before feeding through our final softmax. This is equivalent to replacing equation 5.9 in the Decoder section with the following two equations:

$$\tilde{\mathbf{h}}_t^{\prime a} = \tanh(\mathbf{W}_a[\mathbf{z}_t, \mathbf{h}_t^\prime]) \tag{5.15}$$

$$\mathbf{w}_{t+1} = softmax(\mathbf{W}_p \tilde{\mathbf{h}}_t^{\prime a}) \tag{5.16}$$

where $\mathbf{W}_a \in \mathbb{R}^{(L)\mathrm{x}(A+L)}$ is our attention matrix, and A is our attention vector size.

This attention vector $\mathbf{z}_t$ is calculated as the weighted average over $\{\mathbf{h}_1, ..., \mathbf{h}_k\}$ corresponding to their scores in vector $\mathbf{a}_t = [a_t^1, ..., a_t^k]$ where:

$$a_t^i = \frac{exp(\mathbf{h}_t'^T \mathbf{h}_i)}{\sum_{j=1}^{k} exp(\mathbf{h}_t'^T \mathbf{h}_i)} \tag{5.17}$$

Dropout

We added random dropout layer to the three encoder and three decoder variables during training: $\{\mathbf{x}_j, \mathbf{y}_j, \mathbf{h}_j, \mathbf{h}'_j, \mathbf{c}_j, \mathbf{c}'_j\}$. This meant that $f \in [0, 1]$, fraction of elements that are set to 0 at random from a vector or matrix during training, which has been shown to have a regularising effect (Srivastava et al., 2014). In our case f was a tuned hyperparameter.

Optimisation

We define the words in our true sequence as $\{y_1, ..., y_l\}$, and their one hot representations as $\{\mathbb{I}_1, ..., \mathbb{I}_l\}$. In decoding we calculate distributions for each target word in the sequence $\{\mathbf{w}_1, ..., \mathbf{w}_l\}$. We define our objective function for one point as the sum of cross-entropy losses between the target words in our sequence, (represented as one hot vectors) and the corresponding distributions of those words.

$$\mathcal{L}_{single} = -\sum_{i=1}^{l} \mathbb{I}_i \cdot \log(\mathbf{w}_i) \tag{5.18}$$

For multiple points, we take the average over the minibatch, so the losses dont change dramatically with changing batch size.

$$\mathcal{L}_{mb} = -\frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{l} \mathbb{I}_i \cdot \log(\mathbf{w}_i^j) \tag{5.19}$$

.

We use Adam (Kingma and Ba, 2014) to minimise this objective function. Adam is a stochastic gradient descent mechanism, that uses a momentum based update that adapts during training. It also maintains a separate learning rate for each parameter, and makes adjustments of this based on the gradients and square gradients. This optimiser has proven to be highly effective in training neural networks (Ruder, 2016).

Finally we also apply gradient clipping, where gradients are constrained have a certain magnitude:

$$g' = min(-x, max(g, x)) \tag{5.20}$$

where g is our old gradient, $x \in \mathbb{R}^+$, and g' our clipped gradient. This gradient clipping prevents large shifts occuring in minima next to steep gradients, has been shown to facilitate training (Pascanu et al., 2012).

5.2.3 Code2Vec Decoder Model

Overview

Our Code2Vec Decoder model creates a modified version of the Code2Vec code-vector presented by Alon et al. (2018b), and uses that as intermediate representation for decoding. We differ from Alon et al's paper by building our vector from our argument's *variable path-contexts*, instead whole AST's *path-contexts*. We use the same decoder model as the Seq2Seq (without attention).

We encode our VPCs using a 300-dimensional vector for the path, and 300-dimensional vector for the terminal node, and produce a final code vector of size 300. This is used as both components of our intermediate state, to be decoded our LSTM decoder, which is the same as the Seq2Seq model. We apply a dropout around the intermediate context vectors, and also the inputs, outputs and hidden states of the decoder. We use the same objective function, gradient clipping and optimizer as the Seq2Seq model.

Encoder

In this model our input for each argument is a set of *variable path-contexts*, $\{v_1, ...v_k\}$, where each $v_i = (p_i, n_i)$ and p_i is a path, and n_i is a terminal node. These paths and nodes are encoded simply as integers, indicating items in a vocabulary.

For each v_i we then prepare an embedded context vector $\mathbf{c_i}$ as follows:

$$\mathbf{p}_i = \mathbf{H}^P_{p_i,:} \tag{5.21}$$

$$\mathbf{n}_i = \mathbf{H}^N_{n_i,:} \tag{5.22}$$

$$\tilde{\mathbf{c}_i} = ([\mathbf{p}_i, \mathbf{n}_i])^T \tag{5.23}$$

$$\mathbf{c}_i = \tanh(\mathbf{W_c}\tilde{\mathbf{c}_i} + \mathbf{b_c}) \tag{5.24}$$

Where $\mathbf{H}^P \in \mathbb{R}^{V_p \text{x} D_p}$ is our path embedding matrix, $\mathbf{H}^N \in \mathbb{R}^{V_n \text{x} D_n}$ is our terminal node embedding matrix, $\mathbf{W}_c \in \mathbb{R}^{D_c \text{x}(D_p + D_n)}$, $\mathbf{b}_c \in \mathbb{R}^{D_c}$, V_p is the size of our path vocabulary, V_n is the size of our terminal node vocabulary, D_c is the size of our context vector, D_p is the size of our path embedding vector, D_n is the size of our terminal node embedding vector, and and $\mathbf{X}_{i,:}$ indicates selection of row i from a matrix $\mathbf{X}$.

The final code2vec-style vector $\mathbf{v}$ is calculated as the weighted sum over $\{\mathbf{c}_1, ..., \mathbf{c}_k\}$ corresponding to their scores in vector $\mathbf{a} = [a^1, ..., a^k]$ where:

$$a^i = \frac{exp(\alpha^T \mathbf{c}_i)}{\sum_{j=1}^k exp(\alpha^T \mathbf{c}_i)} \tag{5.25}$$

and $\alpha \in \mathbb{R}^D_c$ is a global attention parameter that is learnt. We use this vector $\mathbf{v}$ to present our final intermediate encoding as :

$$s = (\mathbf{v}, \mathbf{v}) \tag{5.26}$$

Decoder, Optimisation

We use the same decoder as presented in the Seq2Seq model. We also use the same optimiser, objective function, and gradient clipping was presented in the Seq2Seq model.

Dropout

Throughout the network, we added random dropout layers to concatenated context vector and the three decoder variables during training: $\{\tilde{\mathbf{c}}_i, \mathbf{y}_j, \mathbf{h}'_j, \mathbf{c}'_j, \}$ where our dropout fraction was tuned as a hyperparameter.

5.2.4 Code2Vec + Seq2Seq Decoder

Overview

In this model we used both sources of information for our decoder, by concatenating two separate encoded vectors, and passing this through a linear layer. This enabled the model to choose which sections of information were most important. We use the identical set up as the Code2Vec Decoder to generate our code vector, and the identical set up to Seq2Seq for the character encoder and word decoder, and we used the same dropouts, objective function, gradient clipping and optimiser as both.

Encoder

In this model we consider the two encoders above as operators that return tuples. We first the concatenate each component of tuple:

$$s^{s2s} \leftarrow \mathcal{SEQ}(\mathbf{x}) \tag{5.27}$$

$$s^{c2v} \leftarrow \mathcal{COD}(\mathbf{x}) \tag{5.28}$$

$$(\mathbf{s}_1^{comb})^T = [(\mathbf{s}_1^{s2s})^T, (\mathbf{s}_1^{c2v})^T] \tag{5.29}$$

$$(\mathbf{s}_2^{comb})^T = [(\mathbf{s}_2^{s2s})^T, (\mathbf{s}_2^{c2v})^T] \tag{5.30}$$

where $\mathcal{SEQ}$ returns the encoder component of the Seq2Seq model, $\mathcal{COD}$ returns the component of the Code2Vec Decoder, $s^{\mathcal{X}}$ indicates the a tuple of vectors from encoder operation $\mathcal{X}$.

Then each component is fed through a linear layer before combining into the final encoded state to be decoded:

$$\mathbf{s}_1 = \mathbf{W}_d \mathbf{s}_1^{comb} + \mathbf{b}_d \tag{5.31}$$

$$\mathbf{s}_2 = \mathbf{W}_d \mathbf{s}_2^{comb} + \mathbf{b}_d \tag{5.32}$$

$$s = (\mathbf{s_1}, \mathbf{s_2}) \tag{5.33}$$

where $\mathbf{W}_d \in \mathbb{R}^{L\text{x}L_{comb}}$, $\mathbf{b}_d \in \mathbb{R}^L$, L is the decoder size, L_{comb} is the size of the combined concatenated vector.

Decoder, Attention, Dropout, Optimisation

We use the same decoder as presented in the Seq2Seq model, and use Luong Attention in the identical manner. We also use the same optimiser, objective function, and gradient clipping was presented in the Seq2Seq model. We used dropout any variables specified in either the Seq2Seq Model dropout section or the Code2Vec Decoder model dropout section.

5.3 Tokenizations

5.3.1 Tokenizing Signature Data

Since signature data was being investigated character-wise, our encoding vocabulary was short and complete - constrained to 70 characters in all.

If our investigations required just one name, we encoded it as is, and appended a end-of-sequence (<EOS>) token to the list of characters. If we required multiple names, we concatenated the strings, using separator tokens (<S>) in between, which would indicate the type of the previous name. Separators tokens could then be replaced by single characters invalid in a python name, for representation as a string.

---

signature: `def funcA(foo, barA, barB):`
argument: `foo`
tokenization type: variable name + function name + all coarguments
tokenized (sequence): `f,o,o,<S1>,f,u,n,c,A,<S2>,b,a,r,A,<S3>,b,a,r,B,<S3>,<EOS>`
tokenized (string): `"foo|funcA-barA+barB+?"`

Figure 5.1: An example of a tokenization of the signature data. The sequence representation was used for our Seq2Seq, while the string representation was used for our Rote Learner.

5.3.2 Extracting & Tokenizing VPC Data

Although in principle, the VPCs could be taken directly from the entire AST of the accompanying source, a few small modifications were made to facilitate training, and reduce the redundant VPCs in the model. First the AST was pruned to remove terminal nodes that not variables, such as `Load` or `Store` nodes. Such nodes are visible in Figure 2.5.

Although these nodes carry useful information as to whether the target name was being loaded or stored, they almost doubled the number of terminal nodes, and therefore quadratically increase the number of *path-contexts* in the AST. Given our memory and time considerations in both training and preparation we pruned these nodes, noting that the path p for any VPCs eliminated would still be present in the corresponding variable node to which `Load` or `Store` refers. Although the loading and storing information had been removed, the tree still contained a rich information about the relationships between variable - the investigation of Section 4, for instance, was done after this pruning.

Finally, in extracting VPCs we also removed the paths that led to the `arguments` series of nodes, and the string node child of `FunctionDef` . These nodes contained information such as the names of the arguments or name of the function. Since we wanted the function signature data to be investigated separately we removed these paths. Just like Alon et al.

(2018b) we also removed VPCs with paths over a certain length (in our case 10 nodes), again due to resource considerations.

Having extracted our VPCs, we proceded to tokenize them. To do this we took the V_p most common paths and the V_n most common terminal nodes in the training set, and used these as our respective vocabularies. All other paths and nodes were given an <UNK> token in encoding. V_p and V_n were hyperparameters that we maximised given the resources for training.

5.3.3 Tokenizing Descriptions

Finally we tokenized all descriptions by setting them to lower case and using the `nltk` (Bird et al., 2009) Punkt tokenizer (Kiss and Strunk, 2006). We then formed a vocabulary of size V_w from the training set and GloVe embeddings. We did this by taking all words in the training set that had appeared at least 5 times and if they appeared in the GloVe vocabulary, adding them to our training vocabulary. Once this was done, we filled the remaining space in the vocabulary with the most popular words in GloVe that had not already been included. This was done to minimize out-of-vocabulary (<UNK>) tokens. This tokenized training vocabulary was then used for all learners.

5.4 Evaluation

To evaluate our models quantitatively we used the corpus BLEU-4 metric (Papineni et al., 2001), common in machine translation. This is a metric that has been shown to correlate with fluency and adequancy of translations.

To calculate BLEU, we first define a modified ngram precision, p_n as:

$$p_n = \frac{\sum_{ngram \in \mathcal{C}} Count_{clip}(ngram)}{\sum_{ngram' \in \mathcal{C}} Count(ngram')} b$$

where $Count(ngram)$ counts the number of ngrams in the candidate sentence, and $Count_{clip}(ngram)$ counts the number of ngrams in the candidate sentence, up to the maximum number of those ngrams that exist in any reference sentence:

$$Count_{clip}(ngrams) = min(Count(ngrams), MaxRefCount(ngrams))$$

We then define a brevity penalty, to penalise high precision scores, when candidates of length c are much shorter than references of length r.

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases}$$

The sentence-level BLEU-4, is then brevity penalty multiplied by the geometric mean of the modified ngram precisions, up to 4-gram:

$$\text{BLEU-4} = BP \cdot \prod_{n=1}^{4} p_n^{\frac{1}{4}} \tag{5.34}$$

The authors note the geometric can be very harsh on individual sentences, so on a corpus level, we take our modified precisions by summing ngram matches sentence by sentence first.

$$p_n = \frac{\sum_{C \in \{Candidates\}} \sum_{ngram \in \mathcal{C}} Count_{clip}(ngram)}{\sum_{C' \in \{Candidates\}} \sum_{ngram' \in \mathcal{C}'} Count(ngram')}$$

We then calculate our length c and r from the brevity penalty as the total length of the candidates and references respectively. With these modification, the calculation of BLEU-4 is identical

In our experiments we used corpus BLEU-4 for evaluating our models, though occasionally examined sentence BLEU-4 for our analysis. For a qualitative evaluation, we always interrogated the generated descriptions, source code and gold descriptions directly.

5.5 General Procedure

Our general experimental procedure was straightforward. We would take our chosen dataset from Table 4.8 and tokenize it as above. Then we would feed these into our own implementations of models [1], written in Tensorflow (Abadi et al., 2015). Variables in these models were all initialised with Xavier initialisation (Glorot and Bengio). Since our objective function does not correspond well with our evaluation metric, we would train these models on GeForce GTX TITAN X GPUs for a fixed number of epochs, evaluating and saving our models every epoch, and would choose the best model according our to our evaluation metric. This is the approach taken in other papers faced with the same problem (Bahdanau et al., 2014; Barone and Sennrich, 2017). If we were running a number of experiments conjointly all would run for the same number of epochs. In the longest cases (150 epochs with the longest sequences) some models ran for two days.

In the case of our Rote Learner, we would run each model directly on the tokenized data 50 times, to get a sense of the variance according the inherent stochasticity of the model.

[1] https://github.com/condnsdmatters/program_synthesis

Chapter 6

Experiments and Results

6.1 Investigating the Function Signature

6.1.1 Comparing Baseline Models

Experiment Objective

As a baseline, we first wanted to investigate the informative power of simply the name of each argument with regards to different architectures of our model. Since good developer practice often involves giving insightful names to variables[1], we hypothesized that both our Rote Learner and Seq2Seq models should able to generate a plausible description using just the name of the argument. We wanted to use this opportunity to investigate the relative performance of different features of the Seq2Seq model.

Method & Results

Since this experiment would only use the variable name to predict the descrption, we used the **Reduced Random-Split Dataset** listed in Table 4.8. This prevented inflation of BLEU scores due to exact matches being both in the train and validation set.

We ran our Rote Learner model as described in Section 5.5. We then ran our on the

[1]https://github.com/google/styleguide/blob/gh-pages/pyguide.md#s3.16-naming

basic Seq2Seq architectures in an ablation study: we started from a basic seq-to-seq, then adding attention, a bidirectional encoder and finally dropout. These models were trained for a maximum of 150 epochs, in the manner as outline in section 5.5, with our best model on validation set selected. The hyperparameters for these models are found in Appendix Table B.1.

We report our corpus-level BLEU-4 scores in Table 6.1, and confirm our hypothesis that just argument name is a valuable indicator for generating descriptions. We note a strong BLEU-4 score from the Rote Learner of 9.03 ± 0.32 on validation and 10.60 ± 0.30 on test, which demonstrates the informativeness of simple n-character subsequences. This is surpassed by the bidirectional Seq2Seq with attention and dropout, which achieves a BLEU-4 score of 12.68 and 12.72 on validation and test set respectively. In our ablation study we note the improvement that each new feature adds, noting that the pure Seq2Seq model underperforms the Rote Learner, scoring 8.97 and 8.18 on the validation and test set.

Model	BLEU (Validation)	BLEU (Test)
Rote Learner (x50)	9.03 ± 0.32	10.60 ± 0.27
Seq to Seq	8.97	8.18
+ *attention*	10.05	9.89
+ *bidirectional encoder*	11.76	12.34
+ *dropout*	12.68	12.72

Table 6.1: Results of Experiment Comparing Baseline Models

Analysis

We were most surprised by the strength of Rote Learner on this dataset. Given there are no exact matches of (name, description) pairs in this dataset, we were curious to see where the performance had come from. We examined the predictions of the Rote Learner on the validation set by taking their sentence-level BLEU-4 scores, and reading the predictions. We found that although the vast majority evaluate to zero, a number of examples have a coincidental overlap in descriptions, resulting in a non negligible BLEU score. A number of examples also have the same name and almost identical descriptions, giving some predictions very high BLEU scores. A random sample of these non-zero sentence-level BLEU translations are presented in Figure 6.2.

With then examined the Seq2Seq model and ablation study. We were not surprised that the additions each improved the performance of the model in some way. The bidirectionality adds capacity to the model by increasing the size of the hidden vector, and prevents the model forgetting elements at the start of long sequences. The attention allows the model to condition each generated word on the encoder outputs, adding complexity to the model. The dropout also adds a regularizing effect. Each of these should improve the performance of the model.

We were interested to note that the basic Seq2Seq narrowly underperforms the Rote Learner. It is possible that without the contextualising information from attention, the model struggles to learn from long description sequences. In contrast, the Rote Learner can regurgitate these immediately.

Once the attention is added, and the capacity increased, the Seq2Seq's outperformance of the Rote Learner is explanatory. The Seq2Seq synthesises similar (or even the same!) sequences of characters, with different descriptions, into a single description, which may fit better than randomly guessing from a bag of seen sequences.

For instance, the most frequent argument, `name`, occurs in the train set 318 times with different descriptions. It also occurs 52 times in the validation set, again, with different descriptions. For each of these points the Rote Learner predicts a different description from the train set, due to its inherent randomness. On average it scores 0.00646 ± 0.006 in sentence BLEU. The Seq-to-Seq, by contrast, makes the *same* prediction each time, and scores 0.4029, almost two orders of magnitude better. Examining the predicted description of this argument confirms a sensible guess: `name` - *"name of the variable to return."*

In these cases the Seq2Seq model is interpolating in an overdetermined problem, and this highlights one of the advantages of neural networks in machine translation problems.

A Random Sample of Sentence BLEU-4 Scores > 0 from Rote Learner on Random Split

BLEU Score: 0.364 **Confidence**: 100.0%
Argument: `e d g e _ m a t c h`
Description: a function that returns true if the edge attribute dictionary for the pair of nodes (u1 , v1) in g1 and (u2 , v2) in g2 should be considered equal during the isomorphism test . if edge_match is not specified then edge attributes are not considered .
Prediction: a function that returns true if the edge attribute dictionaries for the pair of nodes (<UNK> , <UNK>) in <UNK> and (u2 , <UNK>) in <UNK> should be considered equal during matching .

B: 0.075 **C**: 11.1%
A: `u s e _ l o c k i n g`
D: ‘ bool ‘ . if true use locks for update operation .
P: an optional ‘ bool ‘ . defaults to ‘ true ‘ . an optional bool . defaults to true . if true , the assignment will be protected by a lock ; otherwise the behavior is undefined , but may exhibit less contention .

B: 0.558 **C**: 12.5%
A: `y _ t r u e`
D: ground truth (correct) target values .
P: ground truth (correct) labels .

Table 6.2: A random sample of sentence level BLEU-4 scores that are non-zero on the validation set for a Rote Learner. We note that in the last example, the Rote Learner returns an almost identical description with 12.5%, confidence, and in the first example returns a very similar description with 100% confidence.

6.1.2 Investigating Different Parts of the Signature

Experiment Objective

We noted that in many languages[2] a developer may only have access to the function signature in trying to determine what a variable may be. Therefore quality naming is often important for all parts of the function signature.[3] Having established a naive baseline and the validity of our neural approaches, we decided to investigate how other function signature features could help generate our argument descriptions. We hypothesised that the other names in the function signature could play an important role in contextualising an overdetermined or underdetermined variable name, further improving the performance of the Seq2Seq relative to the Rote Learner.

Method & Results

We investigated how four different tokenizations of function signature data would affect translation behaviour for both our Rote Learner, and our strongest Seq2Seq. These tokenizations were:

1. just the argument name
2. the argument name with the function name
3. the argument name with co-argument names
4. the argument name with both function name and co-argument names

Once again we used the **Reduced Random-Split Dataset**, to prevent advantages due to duplicates, ran tokenization procedures outlined previously, and followed the general procedure outlined in Section 5.5. For our Seq2Seq, we used the best arrangement from our first experiment, including a bidirectional encoder, attention and dropout. We ran these for 150 epochs, and selected the best, just as in Section 6.1.1. The hyperparameters for our trainng are presented in Appendix Table B.2.

[2]such C++, C, C#, which encourage the use of header file + binary, in library development
[3]https://google.github.io/styleguide/cppguide.html#Names_and_Order_of_Includes

We report our corpus level BLEU-4 results in Table 6.3. We note that in all cases the Rote Learner's performance improved with added information, confirming that character overlap in the function signature can be an indicative feature in generating argument description. In it's best case the Rote Learner improved over three points to 12.22 ± 0.23 on validation and 14.34 ± 0.21 on test, thanks to the addition of the function name.

The Seq2Seq model on the other hand benefitted less than the Rote Learner, if at all. In both tokenizations using co-arguments, the model improved by approximately one point. However the tokenization of argument name and function name saw a performance reduction of about 1.5 points, across both validation and test.

Although this agreed with out hypothesis that there is value in extra naming of variables, the underperformance of the Seq2Seq model was surprising and is examined in detail in the analysis section.

Model	BLEU (Validation)	BLEU (Test)
Rote Learner		
name only	9.03 ± 0.32	10.60 ± 0.27
name + function name	12.23 ± 0.23	14.34 ± 0.22
name + co-argument names	12.06 ± 0.16	14.22 ± 0.16
name + function name + co-argument names	11.36 ± 0.14	13.51 ± 0.13
Seq to Seq		
name only	12.46441	12.72522
name + function name	10.77699	11.24894
name + co-argument names	13.69964	13.73890
name + function name + co-argument names	14.13267	13.63351

Table 6.3: Results of the experiments using different parts of the signature

Analysis

In order to probe the Seq2Seq model, we first investigated the predictions qualitatively, reading the descriptions generated on the validation set. We found that many of the errors fell into either the categories of overfitting or nonsense. However, we also found descriptions where the model seemed plausible, with a good degree of fluency. To illustrate this clearly, some typical examples of the model on the *name + co-argument name* model are presented in Table 6.4.

Overfitting
Input: `e n c o d i n g _ t y p e <S1> s e l f <S2> d o c u m e...`
`...n t <S2> r e t r y <S2> t i m e o u t <S2> <END>`
Description: the encoding type used by the api to calculate offsets .
Prediction: the encoding type used by the api to calculate sentence offsets .

Nonsense
I: `i n p u t <S1> n a m e <S2> <END>`
D: a ‘ tensor ‘ of type ‘ complex64 ‘ . a complex64 tensor .
P: ‘ <UNK> ‘ , ‘ <UNK> ‘ . shape is ‘ [... , m , m] ‘ .

Plausible
I: `i n p u t <S1> s t r u c t u r e <S2> m a s k <S2> o u t p...`
`...u t <S2> b o r d e r _ v a l u e <S2> o r i g i n <S2> <END>`
D: binary image to be propagated inside ‘ mask ‘ .
P: binary image where a element is provided and a structuring element .

Table 6.4: Three examples of typical errors in the character Seq-to-Seq model. The first example is an example of overfitting where the whole predicted sentence is found in the training set. The second example is nonsensical, with lots of <UNK> tokens. In the final case, the predicted sentence is original and shows features of the desired sentence.

We then visualized a number of attentions of the translations on the validation set, to investigate what the model was focusing on in these cases. We found that, rather than being dispersed across multiple tokens that might indicate useful phrases in the model, the attention focused overwhelmingly on the final tokens in the RNN in the vast majority of cases. The attentions of the ‘plausible’ and ‘nonsense’ examples from Table 6.4 are visible in Figure 6.1, demonstrating this.

This implies the model is not learning to apply parts of the sequence to parts of the

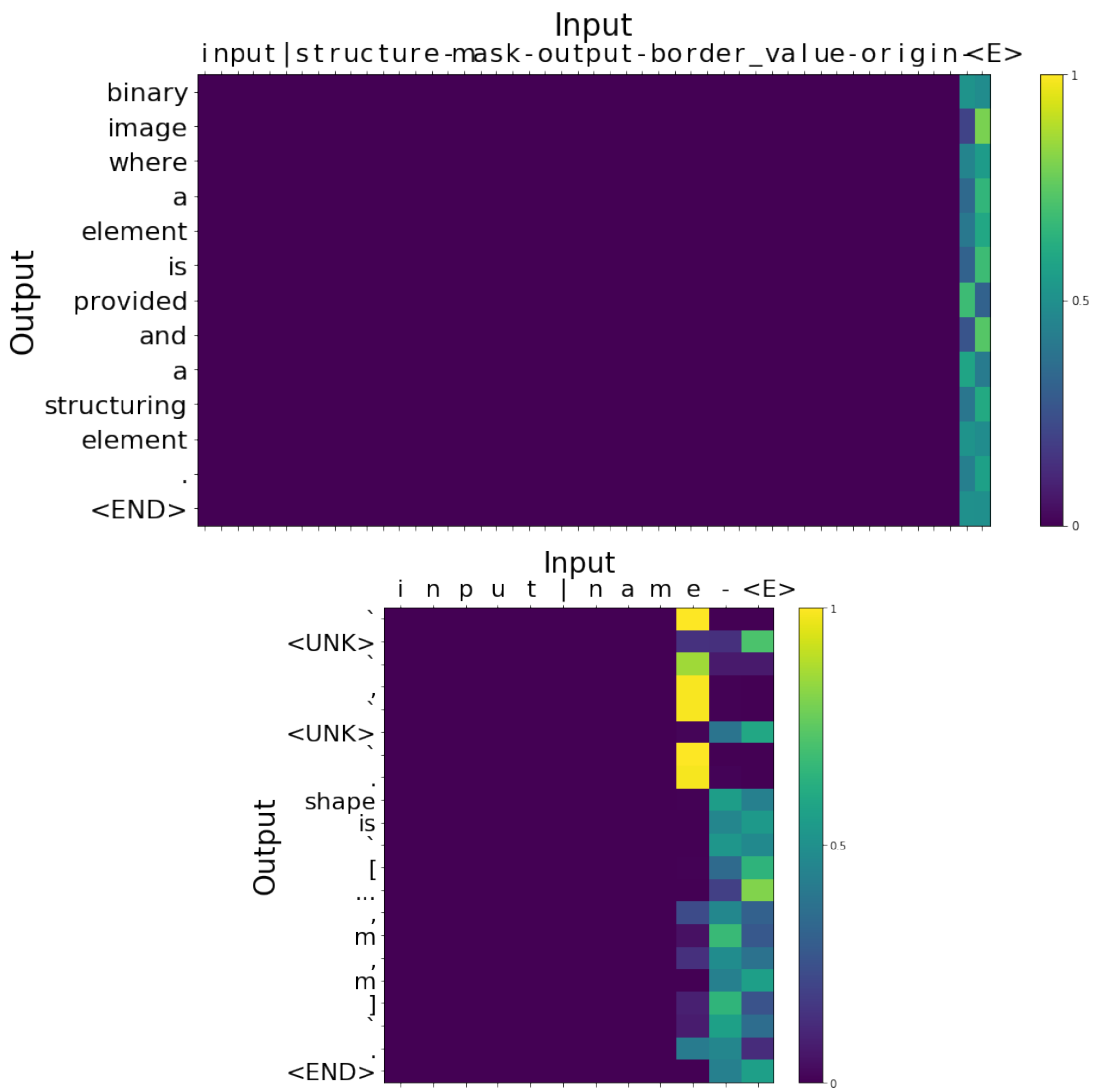


Figure 6.1: An example of some the attentions of from the Name + Other Arguments tokenization on a Seq-to-Seq model, on the Reduced Random-Split Dataset. The top a picture is a typical example from the dataset, where all attention is on the last two tokens which (in this tokenization) never change. The bottom is a less typical example that occurs when the character sequence is very overdetermined.

translation, in the alignment fashion demonstrated by Bahdanau et al. (2014). Instead it appears to be using all the encoded input information to create (almost) the same single vector to contextualise on at each step. This single contextual vector can contain all the input sequence information because, in an RNN, the encoder outputs at step N can contain information from the previous $N-1$ tokens. This in principle allows the model to condition word generation on the all the information up to the final ‘attended’ encoded token.

Although this behaviour was expected for short sequences like variable names, we expected longer sequences (of up to 120 tokens) to be harder to compress, and suspected overcapacity of the LSTM might allow this learning strategy to take place. With less capacity, perhaps the final output of the RNN would not be as informative. It would then be forced to rely on outputs at different points of the sequence - perhaps indicating where the argument name ends, or where an informative sequence of characters lies. However, repeating the experiment with an LSTM encoder with a much smaller hidden state (75 dimensions), resulted in the exactly the same pattern of attention, though with a much reduced BLEU score.

We also repeated the experiment on the larger **Full Random-Split** dataset. In this case the input argument name would be certainly be the most important feature, due to the repetition of points with the same name and description but different signatures. In this case we saw the attention shift from the last token(s) to the separator token just after the variable name. This indicated that the model was learning where the most informative section of the data was, but was still only attending to a single point. This raised the worrying possibility that the network was not even using the encoding intermediate vector, in its decoding in this case.

To disprove this possibility, we needed to check that the rest of the sequence was still being used in decoding. Therefore we took the test sequence and truncated at different points. We found that model did still take into account the rest of the sequence after the separator token, since it generated different translations, but that these all varied along a the same theme of the variable name. This is visible in Figure 6.5. This supported our suggestion that the attention vector was being treated as a contextualising vector in decoding, strongly guiding the type of description generated.

One conclusion to be drawn from this overall investigation is that the attention has not distributed well over characters, failing to pick out individual features. Instead, with these tokenizations, the neural network seems to treat the signature strings like much longer

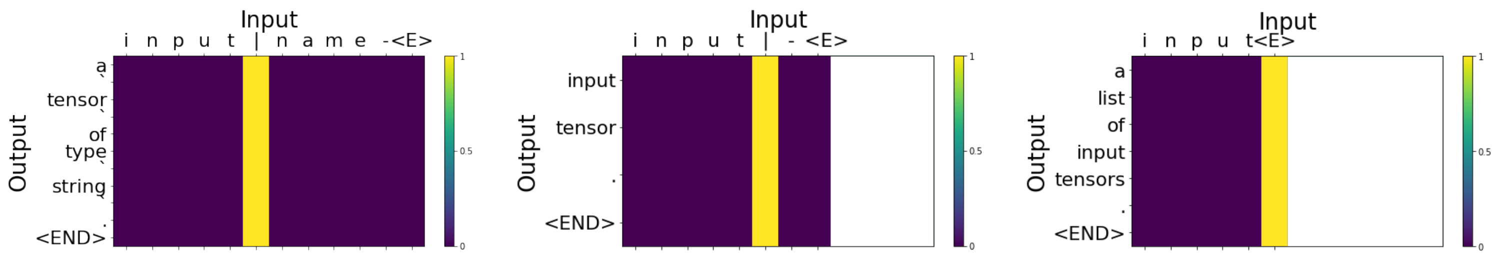


Table 6.5: The attentions from the *name + co-arguments* tokenization on a Seq-to-Seq model trained on the Full Random-Split Dataset. In this dataset, the duplicates of (name, description) pairs make the name the most important feature, hence the attention at the separator token. However the rest of the sequence is still being used by the decoder, as the clipped input sequences still generate different descriptions.

single names. This has led to overfitting in many cases, as the new 'names' are more likely to be unique. In cases such as argument name + function name this seems to lead the network down the wrong path. It also suggests why the Seq2Seq model might not be outperforming the Rote Learner in our corpus-level BLEU score.

However we can also conclude from this experiment that the function signature is capable of providing information to generate documentation - as visibile from the Rote Learner - and that neural methods seem a good way to do this. Unlike our Rote Learner, our Seq2Seq model is capable of generating completely new descriptions, often with good fluency and has numerous examples of doing so. This seems to work best when the Seq2Seq is assimilating information from different training points, instead of overfitting to one. To give an example, Table 6.6 shows the 'plausible' example given above, with all the different training instances that start in the same way - with `input<S1>structure`. Here one can see an original sentence, yet with certain features assimilated from the training, such as the use of *"binary image"*. We note in this example a room for improvement. The potentially helpful input sequence `<S2>mask<S2>` appears to have been ignored by the Seq2Seq, despite appearing relevant to the description. Getting the model to pay attention to these points is the key to our recommendations for improving this type of model.

Further Work

We believe that they key to improving the model is encouraging the model to interrogate the different features of the function signature, whether these be substrings or tokens indicating substring type. As a result we recommend two key areas of investigation as further work for this model: modifications to the attention mechanism and improvements to the tokenization.

Our first suggestion of future work, is that modifications to the attention should be investigated to help it focus on multiple character outputs. One approach could be to replace the peaky softmax with a broader distribution in order to force the model to consider different sections of the data. This could easily be done just be adding a parameter, α to control the entropy of the softmax equation, perhaps varying during during training:

$$softermax(\mathbf{x_i}, \alpha) = \frac{e^{\alpha x_i}}{\sum_i e^{\alpha x_i}} \tag{6.1}$$

A better proposal, however, is to implement a structured attention, which is designed to encourage a sparse focus across contiguous segments (*fusedmax*) or non-contiguous segments (*oscarmax*) of the encoded sequence. (Niculae and Blondel, 2017) This technique has previously been used to improve performance and interpretability in machine translation tasks (Niculae and Blondel, 2017). Another approach, would be to regularise the network by simply penalising it for being too confident in its attention, as has proven to be effective by Pereyra et al. (2017).

A second suggestion of further work is much simpler, involving our tokenization. Since we always tokenized in the same order (argument name, function, coargument) we missed an opportunity for data augmentation, that might help the network pay more attention to the different names in the signature. By adding duplicates with shuffled orders, and maybe even synonyms in descriptions, the model may learn a more generalisable descriptions.

Validation Example
I: `i n p u t <S1> s t r u c t u r e <S2> m a s k <S2> o u t p`...
...`u t <S2> b o r d e r _ v a l u e <S2> o r i g i n <S2>`
D: binary image to be propagated inside ‘ mask ‘ .
P: binary image where a element is provided and a structuring element . <END>

Training Examples starting `i n p u t <S1> s t r u c t u r e`
I: `i n p u t <S1> s t r u c t u r e 1 <S2> s t r u c t u r`...
...`e 2 <S2> o u t p u t <S2> o r i g i n 1 <S2> o r i g i n 2 <S2> <END>`
D: binary image where a pattern is to be detected .

I: `i n p u t <S1> s t r u c t u r e <S2> i t e r a t i o n s <S2> o u t`...
...`p u t <S2> o r i g i n <S2> m a s k <S2> b o r d e r _ v a l u e`
...`<S2> b r u t e _ f o r c e <S2> <END>`
D: binary array_like to be closed . non-zero (true) elements form the subset to be closed .

I: `i n p u t <S1> s t r u c t u r e <S2> o u t p u t <S2> o r`...
...`i g i n <S2> <END>`
D: n-dimensional binary array with holes to be filled

I: `i n p u t <S1> s t r u c t u r e <S2> o u t p u t <S2> <END>`
D: an array-like object to be labeled . any non-zero values in ‘ input ‘ are counted as features...
...and zero values are considered the background .

I: `i n p u t <S1> s t r u c t u r e <S2> i t e r a t i o n s <S2> m a s k`...
...`<S2> o u t p u t <S2> b o r d e r _ v a l u e <S2> o r i g i n <S2>`
...`b r u t e _ f o r c e <S2> <END>`
D: binary image to be eroded . non-zero (true) elements form the subset to be eroded .

I: `i n p u t <S1> s t r u c t u r e <S2> i t e r a t i o n s <S2> m a s k`...
...`<S2> o u t p u t <S2> b o r d e r _ v a l u e <S2> o r i g i n <S2>`
...`b r u t e _ f o r c e <S2> <END>`
D: binary array_like to be dilated . non-zero (true) elements form the subset to be dilated .

Table 6.6: A validation example and all of training points that start with the same sequence of characters: `i n p u t <S1> s t r u c t u r e`. These demonstrate that model is, to some degree, assimilating the information from its training data, in these very long sequences.

6.2 Investigating the Variable Path Contexts

6.2.1 Comparing Code2Vec Decoder to Baselines

Experiment Objective

In our first examinations of the AST data, we aim to validate our initial hypothesis that by using just parts of the code, in the form of VPCs, we could generate reasonable argument descriptions. In particular we wished to assess what possible tokenizations of code paths could prove most effective in doing this, so as to build as strong a Rote Learner model as possible. We could then fully compare this as a strong benchmark against our Code2Vec Decoder model.

Method & Results

For this experiment we used the **Full Random-Split Dataset**, to make the most of the different code paths associated with each argument. We followed the tokenization procedure outlined in Section 5.3.2, using vocabulary size of 15000 for both path and terminal nodes. We then clipped the maximum number of paths per argument to 5000, due to memory constraints.

We then ran our Rote Learner with our standard general procedure, using the four different VPC-only matching criteria, summarised earlier in Table 5.2. These chose a description from a set of training samples in different ways: each either matched full VPCs or VPC subpaths; and each either chose from points proportional to their matches, or simply by taking the most matched points. We ran these different methods to investigate what features might be most indicative for our learner that memorized features, and to find our whether any such features were informative in the first place.

We also ran the Code2Vec Decoder with the same tokenizations and for a maximum of 100 epochs, since we saw little improvement after this point. As usual we selected our model that performed best on our validation set. The hyperparameters for this model are found in Table B.3.

We report our corpus-level BLEU-4 results in Table 6.7. We note that the Rote Learner performed best using the Max Context criterion, which matched points that fully overlapped the most with the VPCs of test. This achieved a reasonable BLEU score of 11.82±0.16 and 12.61±0.12 on validation and test,. However, we noted that the Code2Vec Decoder significantly outperformed this, achieving a BLEU score of 18.94 and 18.13 on validation and test respectively.

These results first confirmed the informative power of VPCs, and suggested that full VPC overlap be an informative feature in describing arguments. It also confirmed the strength of our neural approach to modelling full, and showed great promise for the Code2Vec Decoder as a model.

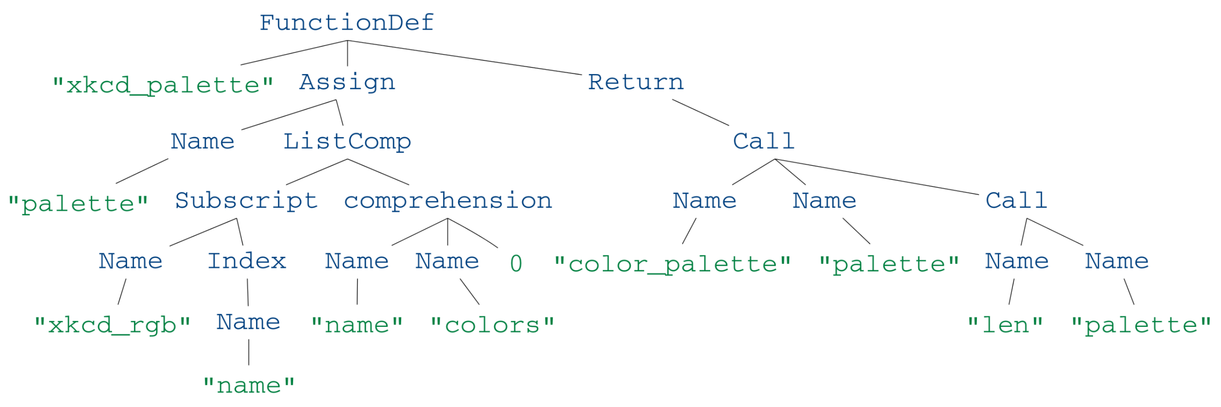
Model	BLEU (Validation)	BLEU (Test)
Rote Learner		
Subpath Proportional	0.71 ± 0.13	0.74 ± 0.07
Subpath Max	6.93 ± 0.34	7.09 ± 0.22
Full Path Proportional	4.94 ± 0.28	5.11 ± 0.18
Full Path Max	11.82 ± 0.16	12.61 ± 0.12
Code2Vec Decoder	18.94	18.13

Table 6.7: Results for Code2Vec Decoder and a series of code only Rote Learners on the Full Random Split Dataset

Analysis

Noting the superior performance of the Code2Vec Decoder model, we first examined the model qualitatively. We sampled points from the validation set, printed their source code and VPCs, and visualizied their attentions as they passed through the model. We observed that although sometimes the model's attention fit overwhelming to a single path, more often than not the model distributed its attention over 4 or 5 paths, even if there were thousands to choose from.

An illustrative example of such analysis is presented in Listings 3 and 4. Listing 3 shows the pruned AST, source code, and the models predictions, while Listing 4 shows



```
def xkcd_palette(colors):
    palette = [xkcd_rgb[name] for name in colors]
    return color_palette(palette, len(palette))
```

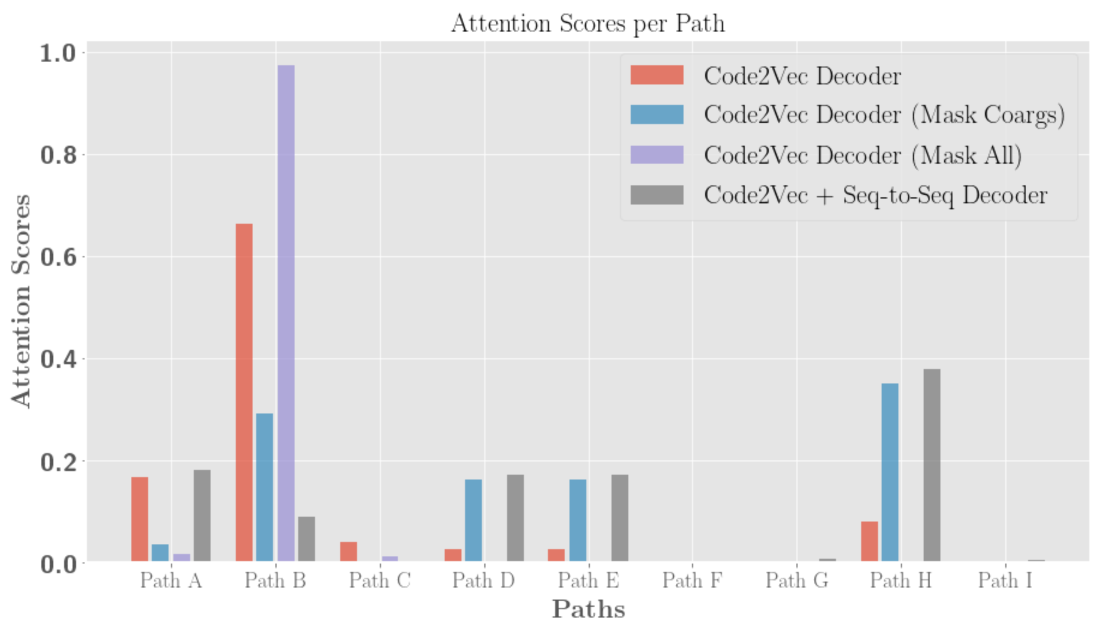
Argument Name: `colors`
Description: list of keys in the “ seaborn.xkcd_rgb “ dictionary .
Prediction: a list of data to read . if none , all other the first will be returned .

Listing 3: A pruned AST from which VPCs were extracted, and its source representation below. Below that the predictions that the Code2Vec Decoder made on the snippet of source. The predicted description correctly identifies a list.

the attented VPCs, and their relative attention scores. From this example it is interesting to note that the model has chosen two paths that indicate list comprehensions feature in the variable, and it has inferred that a list should feature in the description from the code alone.

Having noted that the model seemed to be performing as expected on a qualitative level, we continued to investigate the model more quantitatively. Figure 6.2 shows the distribution of the entropies of the attention in the Code2Vec model, for each point in the validation set. Presented alongside is the distribution of entropies that would have occured if every data point in the validation set had attained a uniform, categorical distribution as attention over its VPC context vectors. The shift to a lower entropy clearly verifies that the Code2Vec's attention focuses on only a fraction of the paths, and, in some few cases (where the entropy is effectively 0) decides resolutely on single points.

We also investigated the sentence-level BLEU-4 scores across the validation set. This



	Paths
Path A	Name ↑ comprehension ↑ ListComp ↓ comprehension : `<UNK>`
Path B	Name ↑ comprehension ↑ ListComp ↑ Assign ↓ Name : `palette`
Path C	Name ↑ comprehension ↓ Name : `name`
Path D	<UNK> : `palette`
Path E	<UNK> : `palette`
Path F	<UNK> : `name`
Path G	<UNK> : `len`
Path H	<UNK> : `color_palette`
Path I	<UNK> : `<UNK>`

Listing 4: The attentions scores for the VPCs corresponding to code in Listing 3, for each of our Code2Vec models. It is interesting that Path B implies that our argument `colors` is involved in a list comprehension and is highly weighted by most models.

metric is prone to low scores (since 4-gram precision of 0 immediately nullifies the score), but gave us a good sense for where the performance improvements come from in the Code2Vec Decoder Model. With this metric we noted both a significant increase in the Code2Vec Decoder's 1.0 scoring sentences (about 12%), and also a similar increase in model's low scoring sentences. This indicates both an improvement that may be down to some form of overfitting (the model is able to find arguments that are very similar, and have similar code paths), but also an improvement due to generating just a few of the correct words, as in the example presented above. A graph presenting the distribution of

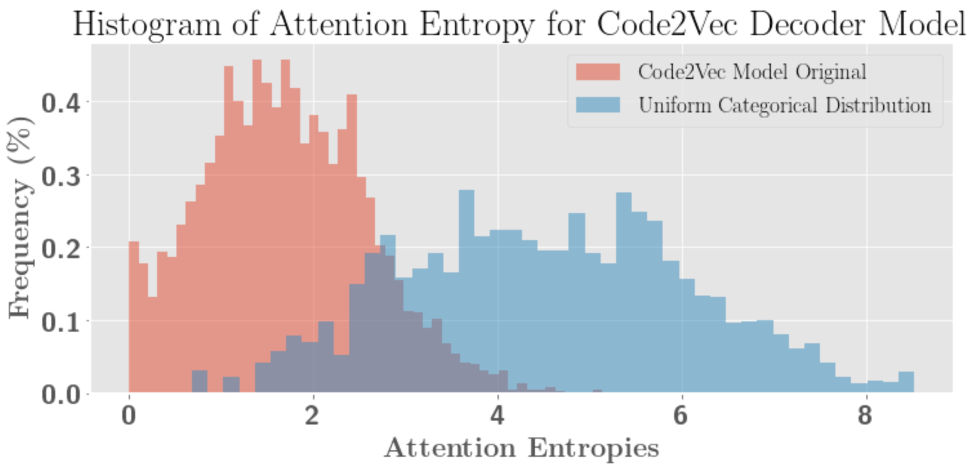


Figure 6.2: The distribution of entropies of the attentions of the Code2Vec Decoder model across the validation set (*red*). For comparison, distributions of entropies of corresponding uniform categorical attentions is also presented (*blue*). This demonstrates that the model choses a small number of paths to attend to, even though the spread of possible paths may be great.

scores in presented in Figure 6.4.

6.2.2 Masking Identifiers in Code2Vec Decoder

Experiment Objective

The Code2Vec Decoder model uses a vocabulary of terminal nodes strings in its VPCs. These terminal nodes strings are almost always object names, which could belong to local variables, built-in functions or even out-of-scope objects. We investigated masking these terminal node names by renaming them, to test the robustness of our model to trivial changes in code structure. We investigated two scenarios: first where the co-arguments are renamed; and second where all the names (including built-in methods) are renamed.

Method & Results

We used the same dataset and tokenization procedure as described in Section 6.2.1 to prepare the tokenized code paths as before. Then we prepared two different masking procedures, to hide the terminal node from the models.

In the first masking, we masked the only the coarguments names in the VPCs. We did this by creating new terminal node identifiers, for ‘first coargument’, ‘second coargument’, and so forth, and replacing the the relevant identifiers in all the VPCs. This meant that the model could still recognise when two code paths belong to the same terminal node, but could not use names to link the coargument to any previously-seen variable.

In our second masking, we repeated this procedure, but provided new terminal nodes for every object in the function: ‘first object’, ‘second object’ and so on. This meant that very indicative global variables were masked, but also any built-in method. The model only had access to enough information to be able to distinguish terminal nodes within a function, it could not generalise to their use between functions.

As before, this model was trained for 100 epochs, with the hyperparameters visible in Table B.4.

We present the results in Table 6.8, noting that Coargument Masking only resulted in a slight performance drop, of one to two BLEU points, from the unmasked variables, while the Full Masking resulted in a significant drop of six BLEU points, but still surpassed the best Rote Learner baseline. We note this as a significant result, demonstrating the robustness of our Code2Vec Decoder, in being able to generate reasonable descriptions, even with every single object in a function renamed.

Model	BLEU (Validation)	BLEU (Test)
Rote Learner (best performer)	11.82 ± 0.16	12.61 ± 0.12
Code2Vec Decoder		
No Masking	18.94	18.13
Coargument Masking	17.02	16.94
Full Masking	13.28	13.07

Table 6.8: Results of investigating the Code2Vec Decoder with masked terminal nodes.

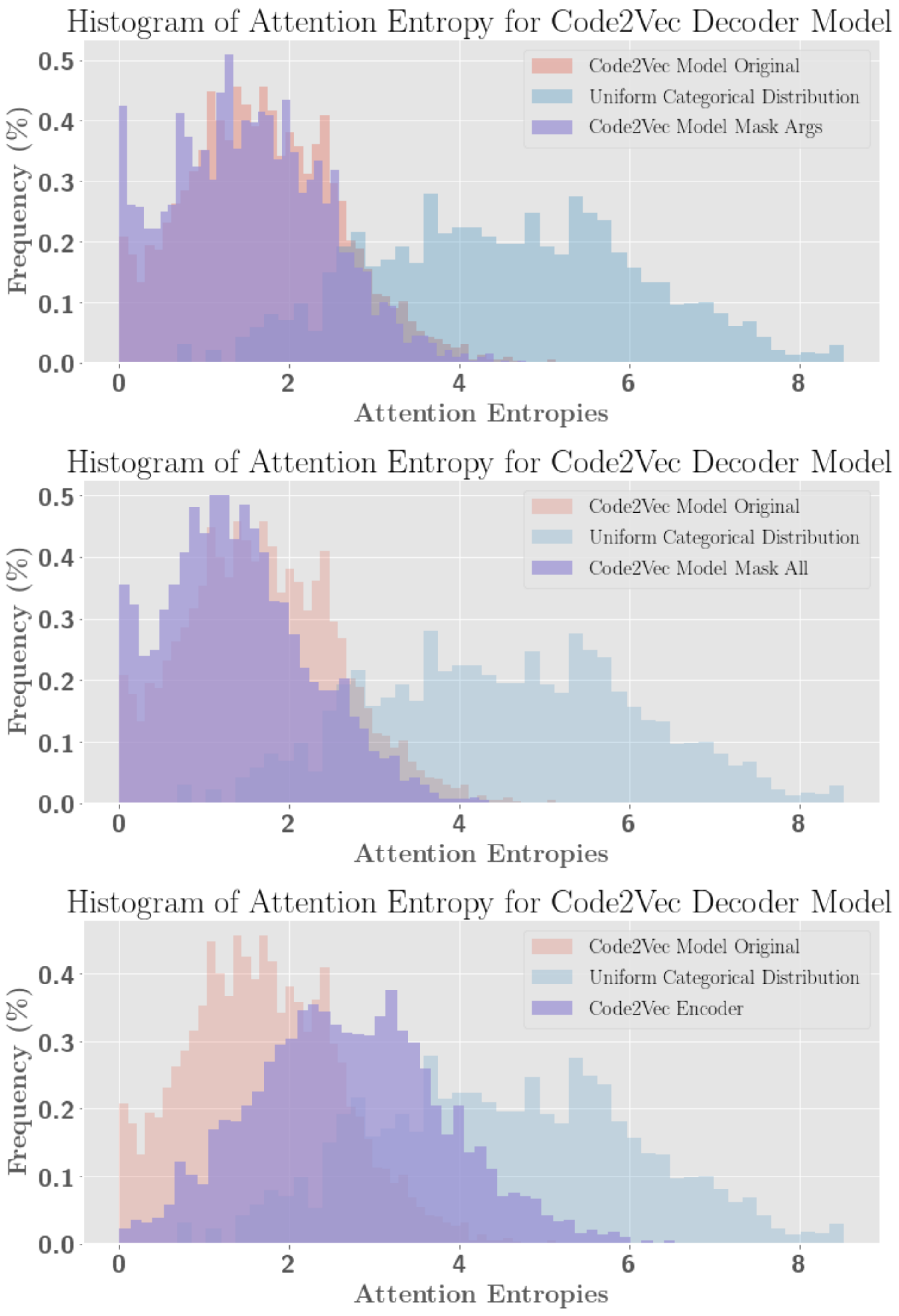


Figure 6.3: The changing distributions of entropies of attentions of the different Code2Vec Decoder variations: Mask Coarguments (*top*), Mask All (*middle*), Code2Vec + Seq2Seq Decoder (*bottom*)

Analysis

To interrogate the models, we once again examined the entropies of both the models' attention on the validation set. These are visualised in Figure 6.3. Masking the coarguments changes the entropy distribution little, except boosting the region of very low entropy. This implies the model behaves similarly in most cases, but has become very sure of some single VPCs in a couple of points. This 'overfitting' to single features might well explain the slight drop in BLEU score.

This behaviour appears accentuated for the fully masked arguments. We see the distribution shift lower, and a greater mass around 0 entropy. Since the terminal nodes have become more generic, the model is more certain of the few paths it recognises, and performance only just surpasses that of the Rote Learner.

Nonetheless the conclusion is remarkable: the model still learns to generate descriptions from code alone, even with all the objects renamed. These objects include built-in methods like `len` or `append` - features that humans may struggle without when attempting the same task. In each of these models the specifics of what the model pays attention to is different - this as can be seen in Figure 4 - but this is a feature representation learning. The model is able to learn the most important features from the VPCs themselves, even if it the features are only sytactical.

6.3 Investigating the Combination of Modalites

6.3.1 Code2Vec + Seq2Seq

Experiment Objective

The Seq2Seq model and Code2Vec Decoder model fundamentally treat code in different ways. The Seq2Seq model investigates code from a naming perspective, learning from character structure of the function signature. The Code2Vec Decoder learns from an AST perspective, picking out patterns in the VPCs. We hypothesized that these two different

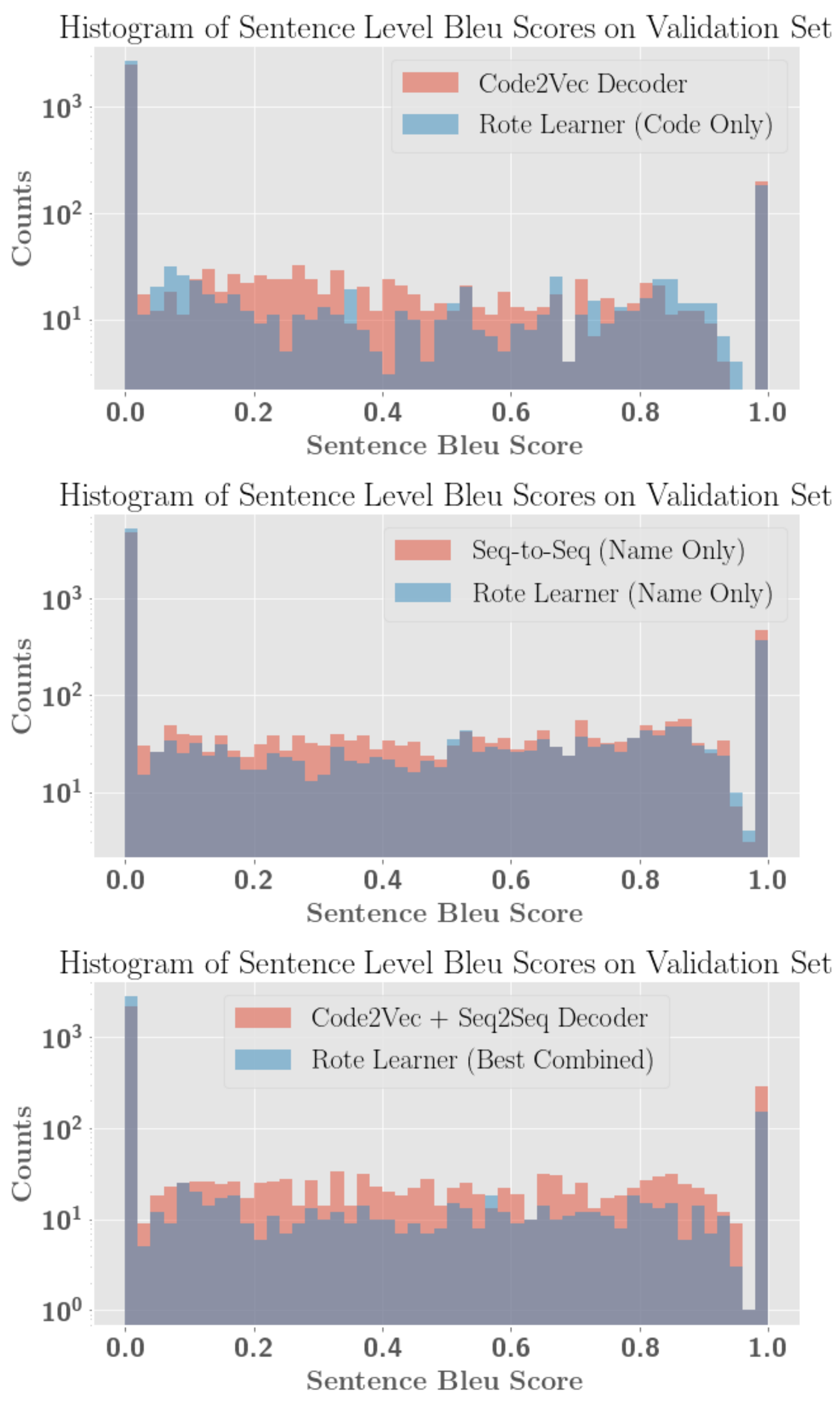


Figure 6.4: A histogram of validation set sentence-BLEU scores for: Code2Vec Decoder (*top*), Seq2Seq (*middle*), Code2Vec + Seq2Seq Decoder (*bottom*). The y-axis is logarithmic.

forms of learning could be successfully combined, to provide a better model than the individual components. In this investigation we compared our individual models to the Code2Vec + Seq2Seq Decoder model, to test this hypothesis.

Method & Results

Given the nature of the AST element of this investigation, we used the **Full Random-Split Dataset** for this investigation, noting this might artificially boost our Seq2Seq only model. We then used the same format of investigation as outlined in previous sections. We ran the all models for 100 epochs, using the name only tokenizations for the Seq2Seq and the unmasked VPCs for the Code2Vec Decoder. The hyperparamters for these model are presented in Appendix Table B.5.

We report our corpus level BLEU score in Table 6.9, and noting that the combined model surpasses both individual models achieving a BLEU of 28.7 and 27.08 on validation and test respectively. This high score is a small improvement on the Seq-to-Seq models, of about 2 points, and at least be partly down to the presence of (name, description) duplicate pairs in the dataset, which would boosts all name-focused methods.

None the less this demonstrates a clear improvement for the combined neural model, that interestingly Rote Learner fails to achieve. The combined Rote Learner performs worse when trying to learn on both modalities, achieving a score almost four points worse than the name only model, which demonstrates one of the strengths of neural methods in combining modalities.

Analysis

Analysis of the combined models largely followed the pattern of those in Sections 6.2. We examined the entropies of the Code2Vec attention distributions and the sentence-level BLEU scores, as evaluated over the whole of the validation set. We found that unsurprisingly the Code2Vec + Seq to Seq Decoder model has a significantly wider distribution of entropies, centered around a higher point, than the individual Decoder. This may partly be

Model	BLEU (Validation)	BLEU (Test)
Rote Learner (code only)	11.84 ± 0.16	12.62 ± 0.12
Code2Vec Decoder	18.94	18.13
Rote Learner (name only)	19.04 ± 0.35	19.69 ± 0.28
Seq2Seq (name only)	26.40	25.40
Rote Learner (combined)	15.02 ± 0.45	15.76 ± 0.30
Code2Vec+ Seq-to-Seq Decoder	28.77	27.68

Table 6.9: Results of the investigation into combining the individual learners.

down to the fact that the model is now also learning from the highly informative variable name, and so depends less heavily on the specific VPCs. Nonetheless it still indicates the model is discerning among codepaths when it makes its decisions.

Although the combined model outperforms the individual models, it appears much of the performance comes from the Seq2Seq part. This can be seen by looking at the individual model BLEU scores in results Table 6.9. These Seq2Seq model's scores are likely inflated, due to the number of duplicates of (name, description) pairs in the dataset, and so this is not the best comparison of Code2Vec Decoder to Seq2Seq. Nonetheless it still demonstrates the primary result of the superior combined model. This model manages to surpass the inflated Seq2Seq, which may benefit from overfitting on certain variable names.

6.4 Investigating the Library Split Dataset

6.4.1 Investigating the Library Split Dataset

Experiment Objective

Our final investigation sought to apply the most difficult test to our models, namely to investigate whether our different models could help generate argument descriptions across completely different code bases and different authors. This involved taking our strongest models and repeating them on our Full Library-Split Dataset.

Method & Results

In this investigation, we repeated the above investigations on the **Full Library Split** dataset. We followed the methodologies outlined above, only making a change to the number of epochs run. In this case we ran for 40 epochs at most, since in all models, the BLEU scores failed to improve very early on. Our hyperparameters for these models are presented in Table B.6, with the for each experiment is reported in Table 6.10.

We report that the neural models significantly struggled to learn the labelled translations on the library split dataset. The weakest models, the Code2Vec Decoders, scored BLEU scores of 0.89 on both train and dev, very close to the code only Rote Learner score of 1.02 ± 0.11, while the argument name models manage to achieve BLEU scores of about 1.5 across both the Rote Learner and Seq-to-Seq models. Finally, the combined model of the Code2Dev + Seq-to-Seq model performed best, by a marginal amount, scoring 1.85 and 1.59 across validation and test sets respectively, outperforming the combined Rote Learner on both modalities by about half a BLEU point.

Model	BLEU (Validation)	BLEU (Test)
Rote Learner (code only)	1.02 ± 0.11	0.85 ± 0.05
Code2Vec		
No Masking	0.89	0.89
Co-argument Masking	0.67	0.67
Full Masking	0.69	0.71
Rote Learner (name only)	1.44 ± 0.17	1.36 ± 0.06
Seq2Seq		
name only	1.53	1.58
name + function name	1.55	1.52
name + co-argument names	1.53	1.45
name + function name +co-argument names	1.51	1.48
Rote Learner (combined)	1.23 ± 0.169	1.157 ± 0.101
Code2Vec + Char to Seq	1.85	1.59

Table 6.10: The results of all models on the Library Split Dataset

Analysis

These results at first glance seem to indicate that the neural models can perform no better in generalised translation than the overfitting Rote Learner models. However these low BLEU scores give little indication to the type of translations that are being generated. To investigate the quality of these translations we examined a random sample of translations from both the Rote Learner and the Code2Vec models. In these we see very little semantic overlap, though the translations make grammatical sense. In particular the Code2Vec descriptions appear shorter and more general than their counterparts with the Rote Learner, but both generally have a poor alignment with the target description. A random sample are shown in Table 6.12.

These low BLEU scores, despite fluent sounding descriptions, lead us to the biggest problem with this task. Generalisation across code different codebases is hard. It requires generalization across code and documentation written by different authors in different contexts with different conventions. Although related work in code language modelling demonstrates that idioms and patterns in code do exist (Allamanis et al.), it has also shown that modelling across project boundaries is significantly harder, as codebases develop idioms and conventions within the project domain (Hindle et al., 2012).

Furthermore the natural language used in descriptions will likely be highly contextualised. Unlike the usage of most words in the real world, descriptions of code artefacts are almost always metaphorical (for instance *"colours"*, instead of *"list of strings"*). and therefore different libraries are under no obligation to use the same language or vocabularies when referring to the same underlying code objects. For instance a list of strings could just as easily by a `x_axis_labels` or `song_categories`, but generating a description surrounding the latter may be difficult without example of the relevant context.

This lack of context and vocabulary, both with code and text, seems to be the case with our tokenized datasets. After tokenization, the Library Split training set description vocabulary is 10% smaller than that of the Random Split training set. To make matters worse, 16% of the Library Split validation description vocabulary then doesn't even appear

	Random Split	Library Split
Tokenized Train Description Vocab Size	5249	4701
Tokenized Valid Description Vocab Size	3012	2866
% of Tokenized Valid Vocab not in Tokenized Train Vocab	**4.1%**	**15.8%**
% of VPCs with <UNK> path elements in Train	37.4%	32.4 %
% of VPCs with <UNK> path elements in Validation	**37.5%**	**57.4%**
% of VPCs with <UNK> terminal nodes in Train	12.3%	8.8%
% of VPCs with <UNK> terminal nodes in Validation	**14.7%**	**44.9%**

Table 6.11: A table showing the change in vocabularies with different Library Splits. The top shows the change in description vocabulary between Random Split and Library Split. This shows that in a Library Split, models learn from less diverse contexts, which overlap less with the validation set. The bottom shows the change in % VPCs with <UNK> code tokens between Random Split and Library Split. The Library Split drastically increases the validation set's <UNK> tokens.

in the training set - a fourfold increase compared to the Random Split case. These facts both indicate that in our tokenized Library Split, the model is learning from less diverse contexts, and yet more often is asked to extrapolate to new contexts.

This behaviour is even worse when it comes to the code paths. Unlike the vocabulary for descriptions, the vocabulary for code paths is set entirely by the training set - there are no Glove preembeddings of paths to help us build a list. The result is that in our tokenized Library Split, the fraction of codepaths with <UNK> path component in the validation set rose from 38% to 58%, and <UNK> terminal nodes rose from 15% to 45%. These large increases further explain why, despite their capacity for abstraction, our Code2Vec models struggled to generalise well. A summary of these results in presented in Table 6.11.

This work shows how challenging it is to generalise code across repository. Syntax trees are vast and diverse (Allamanis et al., 2017a), which poses a problem for vocabularies, even if only taking a small subsection of the tree. Furthermore, the language thats used is necessarily different and metaphorical, which leads to fitting on training that can fail generalise.

Further work on this dataset should focus on different tokenizations and representations that can help capture the similarities betwen the different partitions across libraries, and avoid vocabulary problems. Pointer networks, such as used by Bhoopchand et al. (2016), could be a signficant way of dealing with out-of-vocabulary tokens in code descriptions. So too could generating descriptions at a character level, though this poses its own problems, for particularly long sequences. Similarly different representations of AST codepaths could be investigated, allowing for a more composable version of a VPC.

Argument: `kind`
Description: interpolation mode for the frequency estimator . see :
“ scipy.interpolate.interp1d “ for valid settings .
Code2Vec Decoder: maximum number of iterations to use .
Rote Learner: sample weights .

A: `query`
D: the query parameters , as a dictionary or as an iterable of key-value pairs .
C2V: the name of the <UNK> .
RL: int , or tuple of <UNK> , or tuple of 3 tuples of 2 ints . - if int : the same symmetric cropping is applied to depth , height , and width . - if tuple of 3 ints : interpreted as two different symmetric cropping values for depth , height , and width : ‘ (<UNK> , <UNK> , <UNK>) ‘ . - if tuple of 3 tuples of 2 ints : interpreted as ‘ ((<UNK> , <UNK>) , (<UNK> , <UNK>) , (<UNK> , <UNK>)) ‘

A: `obj`
D: an object .
C2V: a list of tensors to be used .
RL: depth multiplier for <UNK> convolution (also called the resolution multiplier)

A: `networks`
D: list of network names or ids to attach the containers to .
C2V: the number of jobs to use .
RL: copy data from inputs . only affects <UNK> / 2d <UNK> input

A: `G`
D: a networkx graph
C2V: the <UNK> object to use .
RL: ‘ <UNK> ‘ .

Table 6.12: A sample of translations on the Library Split Dataset, in this case with the Code2Vec Decoder model. Although translations have a degree of fluency, the overlap in this case is poor, showing the difficulty of Out-of-Project generalisation.

Chapter 7

Conclusions and Further Work

7.1 Conclusions

At the start of this report we sought to address four questions in our problem formulation:

1. *whether reasonable descriptions can be generated from just lexical names in the function signature?*
2. *whether reasonable descriptions can be generated from the function's abstract syntax tree, without the lexical data?*
3. *whether these models, can be combined in a way that surpasses each individually?*
4. *whether such models can work both in an 'in-project' setting and an 'out-project setting'?*

Over the course of this report we have provided evidence to address answers to each of these questions.

First we conclude that reasonable summaries *can* be generated from just names in the function signature. In particular we have shown that reasonable descriptions can be generated from just the argument name, by looking at its character substructure. Operating on our Reduced Random Split Dataset, our Seq2Seq model generated fluent original descriptions, achieving test BLEU score of 12.72 - surpassing a Rote Learner that regurgitates memorised descriptions based on name similarity. We can also conclude that other aspects of the signature, such as corguments and the function name, also prove informative, since they boost perfomance in our Rote Learner and in most of our Seq2Seq cases. This highlights the informative power of the function signature, and the names that are chosen within it.

Secondly we conclude that reasonable summaries *can* be generated from just the AST, without any reference to the name of the argument. By using the *variable path-context* (VPC) representation, we showed that a Rote Learner that memorised descriptions and matched VPCs could achieve a BLEU score of 12.61 on our Full Random-Split Dataset. This was then significantly outperformed by our Code2Vec Decoder, which achieved a test BLEU score of 18.13, while generating original descriptions. We then demonstrated that this model can still outperform the Rote Learner despite partial or full occlusion of the names of objects in the AST, achieving BLEU scores of 16.94 and 13.07 respectively.

Thirdly we conclude that combining our two different modalities results in a stronger model than either of the two individually. By simply concatenating each encoded vector before decoding, our Code2Vec + Seq2Seq model surpassed individual models by 2 BLEU points on the Full Random-Split Dataset.

Finally our investigation to the last question proves inconclusive, but promising. We notice a significantly worse performance of our models on the ‘out-project’ setting, though they generate sentences with fluency. We demonstrate that due to tokenizations of the dataset, a large fraction of test-set features are rendered out-of-vocabulary in this setting, and combined with a lack of diversity in the training data, this raised the question of whether the models or the tokenizations are the problem in this case.

7.2 Limitations & Critique

In the course of the above investigations, we encountered a number of limitations that affected our progress.

Dataset First and foremost, we noted that the composition of our dataset disproportionately originates from a single source - 41% of our arguments are from `tensorflow`. This proved less problematic for Random Split Dataset but in a Library Split setting this accounts for most of the training data - which also includes `scipy` and `numpy`. Therefore the variance of our training set would have been much reduced, hindering our investigation into generalisation across project.

Secondly, although the size of our Full Datasets are comparable to others[1], our Reduced Datasets are smaller than we desired for our character-level tasks. In particular this small dataset may explain the overfitting that occurred in these tasks. Ideally these sets should have been bigger in the investigations, or subject to data augmentation techniques.

Metric A major limitation of our overall investigation was a lack of automatic metrics for assessing description quality. Without skilled human intervention in reading the source code, it was hard to evaluate whether an argument description is true, even if the n-gram precisions maybe poor. In BLEU evaluations, often multiple synonymous sentences are provided as reference translations in order to improve the validity of n-gram precision. In our case no multiple translations exist. Automating a measure of evaluating descriptions would greatly assist in statistical analysis of where the models fail.

Resources Finally a limitation of our overall investigation was the computational resource available of our hardware. Since we aimed to fit all our work on one GPU, we were constrained to use a limited vocabulary of paths and terminal nodes. Increasing these would likely help in both Random Split and Library Split contexts.

7.3 Further Work

To the best of our knowledge, the work of this report is the first of its kind into the generation of documentation from fine-grained elements of source code. We believe it

[1]such as the Stack Overflow SQL dataset

presents a number of opportunities for future work.

First of all, further work should focus on the use of code's syntactic information, in generating descriptions. Our investigations have highlighted the success this approach, and echo advances made in related tasks of variable naming (Allamanis et al., 2017b; Alon et al., 2018b). We therefore suggest applying other syntactic models to this dataset: semantically annotated graphs (Allamanis et al., 2017b), TreeLSTMs (Tai et al., 2015), or other syntactic models (Yin and Neubig, 2017).

We also think future work should research decomposable representations of AST structures. Alon et al. (2018b) demonstrated this value in their Code2Vec *path-context* model, and we echo it here in our *variable path-context* model. However, our representation of these VPCs as indivisble paths leads to problem of out-of-vocabulary code. By representing VPCs as their decomposed path elements, we may further take advantage of sub patterns in paths while simultaneously addressing this problem. Indeed in recent weeks, a preprint paper has been released by (Alon et al., 2018a) using this approach on a code captioning task - claiming the first use of syntactic information in end-to-end code-to-sequence task.

We also suggest that more work be done on the function signature. The names in this section of code is highly informative, as seen from the proliferation of function-naming tasks (Allamanis et al., 2016, 2017b), and we suggest that advances here could be applied to 'reading' other names in the code. As suggested earlier, investigations here should also focus on improved attention mechanisms (Niculae and Blondel, 2017), or perhaps pointer networks (Vinyals et al., 2015a), to help the networks pay attention to different tokens and include them in the description. Given the success of our combined model, multimodal models should also be investigated.

Finally there is an opportunity for further work in generating larger dataset. We collected data from the list of 300 most popular libraries on PyPI, but our Sphinx plugin could have been applied much more widely. There is still a great deal of well-documented code to be collected - if the right organisations or projects are targeted. Future work could involve collecting data a number of respected organisations on GitHub, as well as all of PyPI.

Appendix A

Dataset Examples

Name	Description	Field	Type
Argument Name	the name of the argument	arg_name	string
Argument Description	the description of the argument	arg_desc	string
Argument Type	the annotated type for the argument	arg_type	string or null
All Arguments	all arguments used in the function	args	list of strings
Other Argument Info	name, desc. and type for all other arguments	arg_info	dict of dicts
Function Name	the name of the function	name	string
Signature	the function signature	sig	string
Path	the full path to the file	filename	string
Library	the library of the function	pkg	string
Source	the source code of the function	src	string
Docstring	the sphinx annotated docstring	docstring	string

Table A.1: Fields collected for each single datapoint. 'Field' refers to the actual key used in yaml file, and 'Type' indicates the type of the data as stored in the file.

Arg. Name (Total)	Top 5 Descriptions	Count
name (1917)	a name for the operation (optional).	1057
	optional op name.	69
	an optional variable_scope name.	41
	a name for this operation (optional).	37
	a string, the name of the layer.	31
x (439)	tensor or variable.	32
	a tensor or variable.	17
	‘bfloat16‘, ‘half‘, ‘float32‘, ‘float64‘, ‘complex64‘, ‘com[...]	14
	numeric ‘tensor‘.	12
	array or sequence containing the data	10
kwargs (303)	additional keyword arguments which will be passed to the ap[...]	12
	optional arguments that “request“ takes.	11
	standard layer keyword arguments.	8
	additional properties to be set on the :class:‘ .google.clo[...]	7
	keyword arguments to pass to base method.	5
axis (263)	axis to broadcast over	14
	the dimensions to reduce. if ‘none‘ (the default), reduces [...]	7
	the index or the name of the axis. 0 is equivalent to none [...]	4
	a ‘tensor‘ of type ‘int64‘.	4
	the axis or axes that were summed.	4
dtype (260)	a ‘tf.dtype‘.	11
	the data type. only floating point types are supported.	8
	default data type for internal matrices. set to np.float32 [...]	7
	the type of the output.	6
	overrides the data type of the result.	6

Table A.2: A table of the most popular names with their respective most popular descriptions and counts of their occurences. These demonstrate a great variety of descriptions despite name repetition.

Variable Name	name		x		kwargs		axis	
Top 5 Libraries	tensorflow	1654	tensorflow	300	tensorflow	62	tensorflow	85
	google	51	matplotlib	43	google	47	pandas	61
	tflearn	50	scipy	35	dask	23	scipy	54
	external	14	tflearn	12	mir_eval	21	dask	20
	absl	13	dask	9	librosa	18	librosa	17

Table A.3: A table of the most popular names with their respective most popular packages and counts of their occurences. These demonstrate a variety of packages generating the similar names.

Appendix B

Model Hyperparameters

The hyperparameters of the best model for each experiment are presented in this section for ease of replication.

Model	**Hyperparameter**	**Value**
-	description vocabulary size	$40,000$
Rote Learner	feature	n-character-gram overlap
Seq to Seq	learning rate	0.001
	batch size	128
	lstm size	300
	max arg. name sequence length	60
	max arg. description sequence length	120
+ *attention*	*attention size*	300
+ *bidirectional encoder*	*bi-lstm size.*	$(300, 300)$
+ *dropout*	*dropout*	0.1

Table B.1: Hyperparameters of Experiment 6.1.1: Comparing Baseline Models

Model	**Hyperparameter**	**Value**
-	description vocabulary size	$40,000$
Rote Learner	feature	n-character-gram overlap
Seq to Seq *all models*	learning rate	0.001
	batch size	128
	bi-lstm size	$(300, 300)$
	attention size	300
	max arg. name sequence length	120
	max arg. description sequence length	120
	dropout	0.1

Table B.2: Hyperparameters of Experiment 6.1.2: Investigating the Function Signature

Model	**Hyperparameter**	**Value**
-	description vocabulary size	$40,000$
	path vocabulary size	$15,000$
	node vocabulary size	$15,000$
	max paths per point	$5,000$
Rote Learner		
	feature	max contexts
	feature	proportional contexts
	feature	max subcontexts
	feature	proortional subcontexts
Code2Vec Decoder	learning rate	0.001
	batch size	128
	dropout	0.1
	max arg. description sequence length	120
	code2vec size (lstm decoder size)	300
	path embedding size	300
	node embedding size	300

Table B.3: Hyperparameters of Experiment 6.2.1: Comparing Code2Vec Decoder to Baselines

Model	Hyperparameter	Value
-	description vocabulary size	$40,000$
	path vocabulary size	$15,000$
	node vocabulary size	$15,000$
	max paths per point	$5,000$
Rote Learner	feature	max contexts
Code2Vec Decoder (all)	learning rate	0.001
	batch size	128
	dropout	0.1
	max arg. description sequence length	120
	code2vec size (lstm decoder size)	300
	path embedding size	300
	node embedding size	300

Table B.4: Hyperparameters of Experiment 6.2.2: Masking Identifiers in Code2Vec Decoder

Model	Hyperparameter	Value
-	description vocabulary size	$40,000$
	path vocabulary size	$15,000$
	node vocabulary size	$15,000$
	max paths per point	$5,000$
Rote Learner		
	feature	n-character-gram overlap
	feature	max contexts
	feature	n-character-gram overlap + max contexts
All Neural Models	learning rate	0.001(*where applicable*)
	batch size	64
	code2vec size	300
	path embedding size	300
	node embedding size	300
	dropout	0.1
	bi-lstm size	$(300, 300)$
	attention size	300
	max arg. name sequence length	120
	max arg. description sequence length	120

Table B.5: Hyperparameters of Experiment 6.3.1: Investigating the Combination of Modalites

Model	**Hyperparameter**	**Value**
-	description vocabulary size	$40,000$
	path vocabulary size	$15,000$
	node vocabulary size	$15,000$
	max paths per point	$5,000$
Rote Learner		
	feature	n-character-gram overlap
	feature	max contexts
	feature	n-character-gram overlap + max contexts
All Neural Models (*where applicable*)	learning rate	0.001
	batch size	64
	code2vec size	300
	path embedding size	300
	node embedding size	300
	dropout	0.3
	bi-lstm size	$(300, 300)$
	attention size	300
	max arg. name sequence length	120
	max arg. description sequence length	120

Table B.6: Hyperparameters of Experiment 6.4.1: Investigating the Library Split

Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Software available from tensorflow.org.

M. Allahyari, S. Pouriyeh, M. Assefi, S. Safaei, E. D. Trippe, J. B. Gutierrez, and K. Kochut. Text Summarization Techniques: A Brief Survey. *arXiv:1707.02268 [cs]*, July 2017.

M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216, San Francisco, CA, USA, May 2013. IEEE. ISBN 978-1-4673-2936-1 978-1-4799-0345-0. doi: 10.1109/MSR.2013.6624029.

M. Allamanis, P. Devanbu, E. T. Barr, and M. Marron. Mining Semantic Loop Idioms from Big Code. page 20.

M. Allamanis, H. Peng, and C. Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. *arXiv:1602.03001 [cs]*, Feb. 2016.

M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A Survey of Machine Learning for Big Code and Naturalness. *arXiv:1709.06182 [cs]*, Sept. 2017a.

M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to Represent Programs with Graphs. *arXiv:1711.00740 [cs]*, Nov. 2017b.

U. Alon, O. Levy, and E. Yahav. Code2seq: Generating Sequences from Structured Representations of Code. *arXiv:1808.01400 [cs, stat]*, Aug. 2018a.

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. Code2vec: Learning Distributed Representations of Code. *arXiv:1803.09473 [cs, stat]*, Mar. 2018b.

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A General Path-Based Representation for Predicting Program Properties. *arXiv:1803.09544 [cs]*, Mar. 2018c.

D. D. J. Arnold. *Machine Translation : An Introductory Guide*. Manchester : NCC Blackwell, 1994. ISBN 185554217X (pbk). Bibliography: p. 219-233.

D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473 [cs, stat]*, Sept. 2014.

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.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, Mar. 1994. ISSN 1045-9227. doi: 10.1109/72.279181.

A. Bessey, D. Engler, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, and S. McPeak. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, Feb. 2010. ISSN 00010782. doi: 10.1145/1646353.1646374.

A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel. Learning Python Code Suggestion with a Sparse Pointer Network. *arXiv:1611.08307 [cs]*, Nov. 2016.

S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python*. O'Reilly, Beijing ; Cambridge [Mass.], 1st ed edition, 2009. ISBN 978-0-596-51649-9. OCLC: ocn301885973.

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003. ISSN 1532-4435.

C. E. Shannon. Prediction and entropy of printed English. *The Bell System Technical Journal*, 30(1):50–64, Jan. 1951. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1951.tb01366.x.

S. F. Chen and J. Goodman. An Empirical Study of Smoothing Techniques for Language Modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL '96, pages 310–318, Santa Cruz, California, 1996. Association for Computational Linguistics. doi: 10.3115/981863.981904.

X. Chen, C. Liu, and D. Song. Tree-to-tree Neural Networks for Program Translation. *arXiv:1802.03691 [cs]*, Feb. 2018.

K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv:1409.1259 [cs, stat]*, Sept. 2014.

J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555 [cs]*, Dec. 2014.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In M. Okada and I. Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435, pages 272–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-77504-1 978-3-540-77505-8. doi: 10.1007/978-3-540-77505-8_23.

S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6): 391–407, Sept. 1990. ISSN 0002-8231, 1097-4571. doi: 10.1002/(SICI)1097-4571(199009) 41:6⟨391::AID-ASI1⟩3.0.CO;2-9.

M. Denkowski and A. Lavie. Meteor Universal: Language Specific Translation Evaluation for Any Target Language. In *Proceedings of the EACL 2014 Workshop on Statistical Machine Translation*, 2014.

L. Dong and M. Lapata. Language to Logical Form with Neural Attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany, 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1004.

R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431, San Francisco, CA, USA, May 2013. IEEE. ISBN 978-1-4673-3076-3 978-1-4673-3073-2. doi: 10.1109/ICSE.2013.6606588.

J. Firth. A Synopsis of Linguistic Theory 1930-1955. In *Studies in Linguistic Analysis.* Philological Society, Oxford, 1957. reprinted in Palmer, F. (ed. 1968) Selected Papers of J. R. Firth, Longman, Harlow.

B. Floyd, T. Santander, and W. Weimer. Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186, May 2017. doi: 10.1109/ ICSE.2017.24.

M. Freitag and Y. Al-Onaizan. Beam Search Strategies for Neural Machine Translation. *arXiv:1702.01806 [cs]*, Feb. 2017.

M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE '10*, page 147, Santa Fe, New Mexico, USA, 2010. ACM Press. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882315.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. page 8.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0-262-03561-8 978-0-262-03561-3.

Z. S. Harris. Distributional Structure. WORD, 10(2-3):146–162, Aug. 1954. ISSN 0043-7956, 2373-5112. doi: 10.1080/00437956.1954.11659520.

A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Zurich, Switzerland, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1. chapter Distributed Representations, pages 77–109. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8): 1735–1780, Nov. 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735.

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.

A. Karpathy and L. Fei-Fei. Deep Visual-Semantic Alignments for Generating Image Descriptions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3128–3137, 2015.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

T. Kiss and J. Strunk. Unsupervised Multilingual Sentence Boundary Detection. *Comput. Linguist.*, 32(4):485–525, Dec. 2006. ISSN 0891-2017. doi: 10.1162/coli.2006.32.4.485.

D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, Jan. 1984. ISSN 0010-4620. doi: 10.1093/comjnl/27.2.97.

P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. Moses: Open Source Toolkit for Statistical Machine Translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, pages 177–180, Prague, Czech Republic, 2007. Association for Computational Linguistics.

Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow. Gated Graph Sequence Neural Networks. In *Proceedings of ICLR'16*, Apr. 2016.

Z. C. Lipton, J. Berkowitz, and C. Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv:1506.00019 [cs]*, May 2015.

S. Liu, N. Yang, M. Li, and M. Zhou. *A Recursive Recurrent Neural Network for Statistical Machine Translation*, volume 1. June 2014. doi: 10.3115/v1/P14-1140.

A. Lopez. Statistical machine translation. *ACM Computing Surveys*, 40(3):1–49, Aug. 2008. ISSN 03600300. doi: 10.1145/1380584.1380586.

P. Loyola, E. Marrese-Taylor, and Y. Matsuo. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. *arXiv:1704.04856 [cs]*, Apr. 2017.

M.-T. Luong, H. Pham, and C. D. Manning. Effective Approaches to Attention-based Neural Machine Translation. *arXiv:1508.04025 [cs]*, Aug. 2015.

C. J. Maddison and D. Tarlow. Structured Generative Models of Natural Source Code. *arXiv:1401.0514 [cs, stat]*, Jan. 2014.

G. Melis, C. Dyer, and P. Blunsom. On the State of the Art of Evaluation in Neural Language Models. *arXiv:1707.05589 [cs]*, July 2017.

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]*, Jan. 2013a.

T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed Representations of Words and Phrases and their Compositionality. *arXiv:1310.4546 [cs, stat]*, Oct. 2013b.

D. Movshovitz-Attias and W. W. Cohen. Natural Language Models for Predicting Programming Comments. page 6.

T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 532, Saint Petersburg, Russia, 2013. ACM Press. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491458.

V. Niculae and M. Blondel. A Regularized Framework for Sparse and Structured Neural Attention. *arXiv:1705.07704 [cs, stat]*, May 2017.

M. A. Nielsen. Neural Networks and Deep Learning. 2018.

K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*, page 311, Philadelphia, Pennsylvania, 2001. Association for Computational Linguistics. doi: 10.3115/1073083.1073135.

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training Recurrent Neural Networks. *arXiv:1211.5063 [cs]*, Nov. 2012.

J. Pennington, R. Socher, and C. Manning. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162.

G. Pereyra, G. Tucker, J. Chorowski, L. Kaiser, and G. Hinton. REGULARIZING NEURAL NETWORKS BY PENALIZING CONFIDENT OUTPUT DISTRIBUTIONS. page 11, 2017.

R. J. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1(2):270–280, June 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.2.270.

R. Kneser and H. Ney. Improved backing-off for M-gram language modeling. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 181–184 vol.1, May 1995. ISBN 1520-6149. doi: 10.1109/ICASSP.1995.479394.

V. Raychev, P. Bielik, and M. Vechev. Probabilistic Model for Code with Decision Trees. page 17.

S. Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*, Sept. 2016.

A. M. Rush, S. Chopra, and J. Weston. A Neural Attention Model for Abstractive Sentence Summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 379–389, Lisbon, Portugal, 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1044.

G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. [NOT NATURALNESS] Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering - ASE '10*, page 43, Antwerp, Belgium, 2010. ACM Press. ISBN 978-1-4503-0116-9. doi: 10.1145/1858996.1859006.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. page 30.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. *arXiv:1409.3215 [cs]*, Sept. 2014.

T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531, May 2011. ISBN 2379-190X. doi: 10.1109/ICASSP.2011.5947611.

K. S. Tai, R. Socher, and C. D. Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, Beijing, China, 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1150.

Z. Tu, Z. Su, and P. Devanbu. On the Localness of Software. page 12.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. *arXiv:1706.03762 [cs]*, June 2017.

O. Vinyals, M. Fortunato, and N. Jaitly. Pointer Networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015a.

O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3156–3164, 2015b.

M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345, Florence, Italy, May 2015. IEEE. ISBN 978-0-7695-5594-2. doi: 10.1109/MSR.2015.38.

K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *arXiv:1502.03044 [cs]*, Feb. 2015.

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, 9. doi: 10.1109/ASE.2015.36.

P. Yin and G. Neubig. A Syntactic Neural Model for General-Purpose Code Generation. *arXiv:1704.01696 [cs]*, Apr. 2017.

T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent Trends in Deep Learning Based Natural Language Processing. *arXiv:1708.02709 [cs]*, Aug. 2017.

V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv:1709.00103 [cs]*, Aug. 2017.