UNIVERSITY OF CALIFORNIA

Los Angeles

Encoding Abstract Syntax Trees (AST) via distance based self-attention mechanism

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Electrical & Computer Engineering

by

Rohan Dutta

2021

ABSTRACT OF THE THESIS

Encoding Abstract Syntax Trees (AST) via distance based self-attention mechanism

by

Rohan Dutta
Master of Science in Electrical & Computer Engineering
University of California, Los Angeles, 2021
Professor Jonathan Chau-Yan Kao, Chair

Code summarization and generation are valuable tasks to master for their wide range of applications in code readability and code translation to name a few. This research work is an extension of previously conducted research on the use of PLBART, a sequence-to-sequence transformer model used for a variety of program and language understanding and generation (PLUG) tasks. The ultimate goal is to improve the performance of PLBART by modifying the noise function of it's denoising autoencoder. The current noise function corrupts code tokens randomly, but we hope to improve performance by masking nodes on the corresponding Abstract Syntax Tree (AST) instead.To integrate the AST structure into the self-attention mechanism, we adopt the dependency-guided self-attention mechanism explored in NLP literature in particular [ZKC21]. However, from the AST, we cannot compute distances between all tokens that appear in a code since they need not necessarily appear in the parse tree structure. So, we investigate how we can derive distances between tokens from the AST structure.

The thesis of Rohan Dutta is approved.

University of California, Los Angeles

2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to sincerely thank all professors on my thesis committee: Prof. Jonathan Kao, Prof. Kai-Wei Chang and Prof. Corey Arnold for their mentorship, support and excellent guidance. Additionally, this thesis research wouldn't be possible without my PhD candidate mentor Mr. Wasi Ahmad. His patience, support and guidance throughout this project was invaluable and additionally I truly appreciate him providing me the much needed resources and advice on how to learn the important details which were required to complete this project.

CHAPTER 1

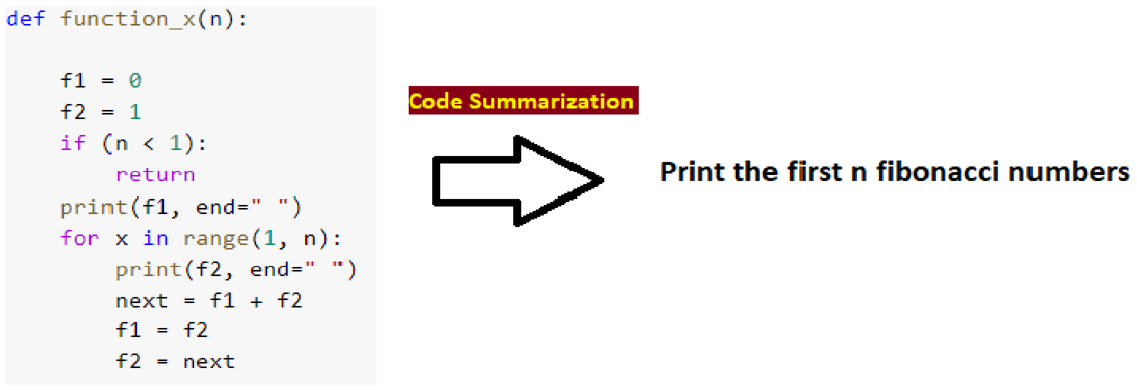
Introduction

1.1 Code Summarization and generation

The premise of this research is to be able to improve performance in Code Summarization and generation tasks. The idea is to train a Transformer model to be able to summarize programming language i.e. Code Summarization (Programming Language to Natural Language). A simple example in *Fig 1.1*

Code Generation is essentially the reverse process of Code Summarization. Being able to generate code from natural language. A simple example would look like *Fig 1.2*

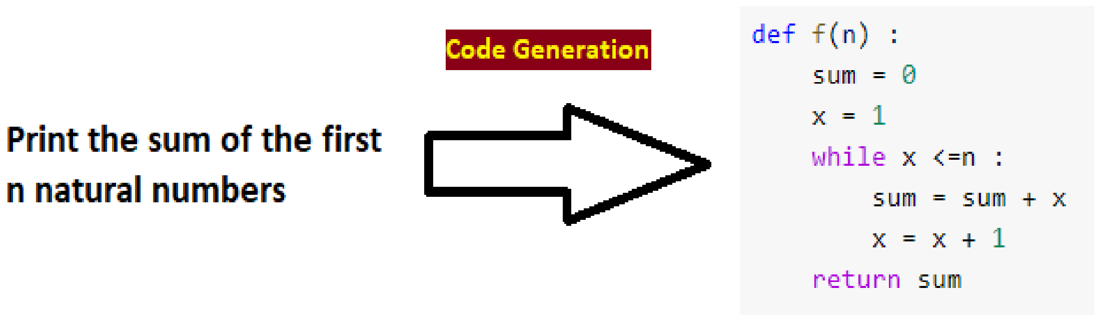
Note that both examples use Python3 as the source or destination programming language and English as the source or destination natural language. Hence training the transformer model can be extended to many different programming languages such as Java, Ruby, JS,



```
def function_x(n):

    f1 = 0
    f2 = 1
    if (n < 1):
        return
    print(f1, end=" ")
    for x in range(1, n):
        print(f2, end=" ")
        next = f1 + f2
        f1 = f2
        f2 = next
```

Figure 1.1: Code Summarization



```
def f(n) :
    sum = 0
    x = 1
    while x <=n :
        sum = sum + x
        x = x + 1
    return sum
```

Figure 1.2: Code Generation

$$PL_{L_1} \xrightarrow[\text{Code Summarization}]{} NL \xrightarrow[\text{Code Generation}]{} PL_{L_2}$$

Figure 1.3: Code Translation

PHP, Go, etc. together. So, essentially this is can be formulated as a multilingual multi-task learning problem. Most research in this field restricts the natural language to English due to easy availability of data in English in most predominant data sources such as GitHub, Stack Overflow, etc.

The advantages of perfecting such a multilingual multitask model has many apparent advantages. For starters it would be easy for programmers to summarize code without having to read through the code and this could help speed up many pipelines in industry.

We would also be able to perfect code translation as a direct application of this for example we could translate Programming Language (PL) from Language 1 (L1) to Language (L2) using Natural Language as an intermediary. See *Fig 1.3*

Such a process would allow translation between languages such as Java and Python, vice versa, and so on. In short the advantages of being able to achieve better performances in Code Summarization and Generation are manifold and we attempt to achieve improvements in the same.

1.2 Denoising Autoencoder

Autoencoders are Neural Networks which are commonly used for feature selection and extraction. However, when there are more nodes in the hidden layer than there are inputs, the Network is risking to learn the so-called "Identity Function" meaning that the output equals the input, marking the Autoencoder useless. Denoising Autoencoders solve this problem by corrupting the data on purpose by randomly nulling some of the input values. [Mon17]

1.3 PLBART

This research work is an extension of the PLBART paper [ACR21]. PLBART is a sequence-to-sequence model capable of performing a broad spectrum of program and language understanding and generation tasks (PLUG). PLBART is trained on code summarization: Programming Language to Natural Language (PL $\rightarrow$NL) and generation (NL $\rightarrow$PL) on many languages (e.g., Java, Python, Ruby, JS, PHP, Go, etc.) together via denoising autoencoding. Denoising autoencoding *(details in 1.2)* involves reconstructing an input text that is corrupted by a noise function. This forces the model to learn language syntax and semantics. Different noising strategies may include token masking, token deletion and token infilling.

PLBART rivals or outperforms existing state-of-the-art models in seven programming languages. Also, further analysis reveals that PLBART learns program syntax, style, logical flow and other features that are crucial to program. However, PLBART was found to have a very small improvement on summarization tasks due to multilingual multi-task modeling. Furthermore, training models on languages grouped by their characteristics, didn't provide any significant benefit on the downstream tasks. An example of languages grouped by their characteristics are as follows:
- Compiled (Java, Go, Ruby) vs Interpreted (Python, PHP, JS)
- Strong typed (Java, Go, Ruby, Python) vs Weak typed (PHP, JS)
- Static (Java, Go) vs Dynamic (Ruby, Python, PHP, JS)

1.4 Abstract Syntax Trees (AST)

1.4.1 Syntax trees

A syntax tree in linguistics is a tree that represents the syntactic structure of simple English sentences and phrases. For example the simple sentence “the man plays soccer” can be broken down into a noun phrase (NP) and a verb Phrase (VP) and so on, as in *Fig 1.4*.

1.4.2 What are AST’s?

Like syntax trees in linguistics, an abstract syntax tree (AST) is a way of representing the syntax of a programming language as a hierarchical tree-like structure. ASTs are widely used in compilers to check code for accuracy. If the generated tree contains errors, the compiler prints an error message. ASTs are highly specific to programming languages, but research is underway on universal syntax trees. Examples of Java and Python AST’s are in *Fig 1.5* and *Fig 1.6* respectively

1.4.3 Why AST’s?

Manipulating text is dangerous in code; it shows the least amount of context. Ex: trying to manipulate text using string replacements or regular expressions. Even manipulating tokens aren’t easy. While we might know what a variable is, if we would want to rename it, we would have no insight into things like the scope of the variable or any variables it might clash with. The AST provides enough information about the structure of code that we can modify it with more confidence. We could for example determine where a variable is declared and know exactly which part of the program this affects due to the tree structure. The four steps of code modification using AST: Parsing, Traversing, Modify, Generate.

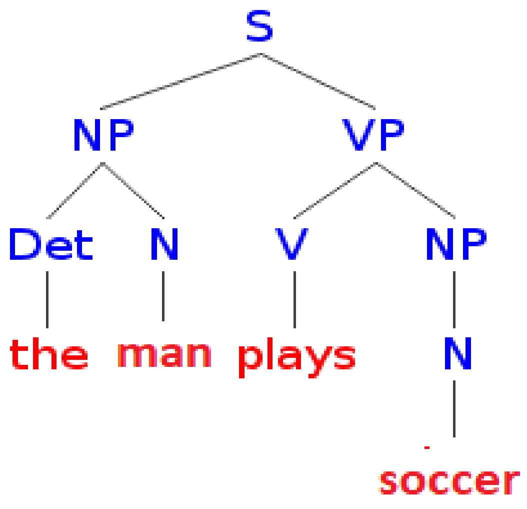


Figure 1.4: Syntax Trees in Lingustics

1.4.4 How AST's work on code?

A tokenizer splits the input stream representing the expression into a list of tokens, and a parser which takes the list of tokens and constructs a parse tree or ast from it. So, the expression $1 + 2 * (3 + 4)$ might be split into a list of tokens like this:

1 - int

+ - add_operator

2 - int

$*$ - mul_operator

(- lparen

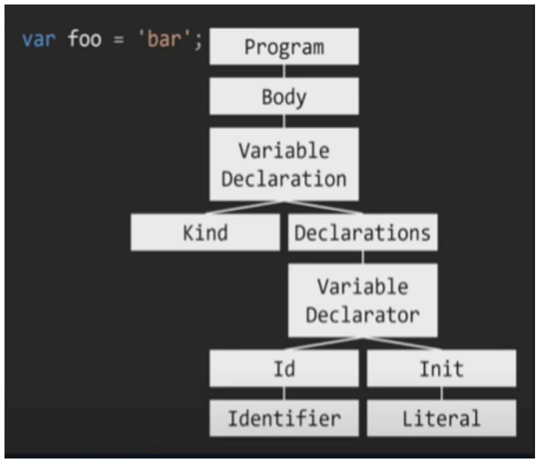
3 - int

+ - add_operator

4 - int

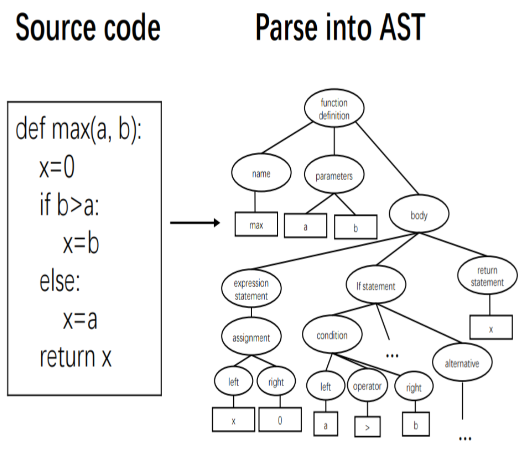
) - rparen

The first column is the actual text value. The second represents the token type. These



```
var foo = 'bar';
```

Figure 1.5: AST in Java



```
def max(a, b):
    x=0
    if b>a:
        x=b
    else:
        x=a
    return x
```

Figure 1.6: AST example Python *source:* [GRL21]

tokens are fed into the parser, which is built from grammar and recognizes the tokens and builds the parse tree.

CHAPTER 2

Related Works

2.1 GraphCodeBERT: Pre-Training code representations with data flow [GRL21]

The paper leverages semantic-level information of code, i.e. data flow, for pretraining. Data flow is a graph, in which nodes represent variables and edges represent the relation of "where-the-value-comes-from" between variables. Compared with AST, data flow is less complex and does not bring an unnecessarily deep hierarchy, the property of which makes the model more efficient.

The model takes source code paired with comments and the corresponding data flow as the input and is pre-trained using standard masked language modeling. It is trained with two structure-aware tasks:

- predict where a variable is identified from
- data flow edges prediction between variables

In summary, the contributions of this paper are:

(1) GraphCodeBERT is the first pre-trained model that leverages semantic structure of code to learn code representation.

(2) Introduction of two new structure-aware pre-training tasks for learning representation from source code and data flow.

(3) GraphCodeBERT provides significant improvement on four downstream tasks, i.e. code search, clone detection, code translation, and code refinement.

2.2 PLBART: Unified Pre-training for Program Understanding and Generation [ACR21]

This paper introduces PLBART, a sequence-to-sequence model capable of performing a broad spectrum of program and language understanding and generation tasks (PLUG). It uses denoising sequence-to-sequence pretraining to utilize unlabeled data in PL and NL. Such pre-training lets PLBART reason about language syntax and semantics. It achieves state of the art performance in software engineering tasks, including code summarization, code generation, and code translation. *(Discussed in detail in 1.3)*

2.3 Project CodeNet: Simplified Parse Tree [PKJ21]

Introduced "Project CodeNet", a first-of-its-kind, very large scale, diverse, and high-quality dataset to accelerate the algorithmic advancements in AI for Code. The dataset consists of 14M code samples and about 500M lines of code in 55 different programming languages, sample input and output test sets for over 7M code samples. The coding tasks include code similarity, classification for advances in code recommendation algorithms, and code translation between a large variety programming language, to advances in code performance (both runtime, and memory) improvement techniques. Additionally: several preprocessing tools in Project CodeNet can be used to transform source codes into representations that can be readily used as inputs into machine learning models. Ex: SPT (simplified Parse Tree)

2.3.1 SPT-Generator Batch Processing

Employed the SPT generator tool to AI4Code SPT generator Featurize programming source code for the ML/DL pipeline in AI4Code project which currently supports 4 languages (C, C++, Python, Java). This tool generates Simplified Parse Tree (SPT) for each recognized program, which follows the similar idea presented in Facebook Aroma paper [LYB19].

2.4 Language-Agnostic Representation learning of source code from structure and context [ZKC21]

This paper introduces the Code Transformer model that integrates both context and structure into it's self-attention mechanism. Self-attention is the core operation powering the transformer [VSP17] to focus on relevant parts of the input. The source code provides the context and it's corresponding Abstract Syntax Tree (AST) provides the structure. The AST is a complementary representation of the same computer program discussed in Chapter 1.4. The Code Transformer model uses language agnostic features of the source code and AST and can hence be extended to multiple languages. This paper achieves state of the art on code summarization in 5 languages.

2.5 Code and Named Entity Recognition in StackOverflow [ACR20]

This paper introduces a new named entity recognition (NER) corpus for the computer programming domain, consisting of 15,372 sentences annotated with 20 fine-grained entity types. Indomain BERT [DCL19] representations (BERTOverflow) were trained on 152 million sentences from StackOverflow, which lead to an absolute increase of +10 F1 score over off-the-shelf BERT.The paper also presens the SoftNER model which achieves an overall 79.10 F1 score for codeand named entity recognition on StackOverflow data. The model incorporates a context-independent code token classifier with corpus-level features to improve the BERT based tagging mode

CHAPTER 3

Methodology

In this chapter we discuss the pipeline we use to obtain token mappings from the raw code/method snippets downloaded in our datasets (described in Chapter 4). We make use of the stage1 preprocessing pipeline used in [ZKC21] which consists of several steps:

- (1) Tokenizing the code snippet
- (2) Apply snippet-level filtering (removing comments, masking strings, etc)
- (3) Parsing the AST from the code snippet (this takes the biggest amount of time)
- (4) Transform the AST parsing result into a graph
- (5) Calculate mapping between tokens and graph nodes

3.1 Textual Code Snippet Preprocessing

Textual code snippet preprocessing consists of both Step 1: Tokenizing the code snippet and Step 2: Applying snippet level filtering. We tokenize code snippets with a language specific tokenizer: Pygments. Further preprocessing includes removing comments, removing empty lines. Hard coded string and numbers are replaced with a special *"mask string"* and *"mask number"*. Indentation style is detected and replaced with a corresponging *indent* or *dedent* token. Tokens are made into sub tokens of 5 and if a token consists of less than 5 sub tokens, the remaining spaces are filled with a special [PAD] token. Any remaining tokens that only consist of white spaces are removed. The only white space characters that are kept are line breaks. Finally, code snippets where the Pygments tokenizer cannot parse a token are discarded.

3.2 Parsing the AST

To obtain language-specific AST's we make use of AST parser from the java-parser project for java code and for Python, JavaScript, Ruby and Go, we use semantic. Snippets that lead to an AST parse error are discarded.

3.2.1 Semantic

Semantic [Git] is a Haskell library and command line tool for parsing, analyzing, and comparing source code.The first stage of our preprocessing pipeline makes use of semantic to generate ASTs from code snippet that are written in Python, JavaScript, Ruby or Go. Semantic is capable of parsing source code in a variety of languages. The generated ASTs mostly share a common set of node types which is important for multilingual experiments such as this one. Unfortunately semantic does not Java and hence a separate AST parser must be employed. To obtain the ASTs, we rely on the –json-graph option that has been dropped temporarily from semantic. As such, the stage 1 preprocessing requires a semantic executable built from a revision before Mar 27, 2020.

3.2.2 JavaParser and Java Method Extractor

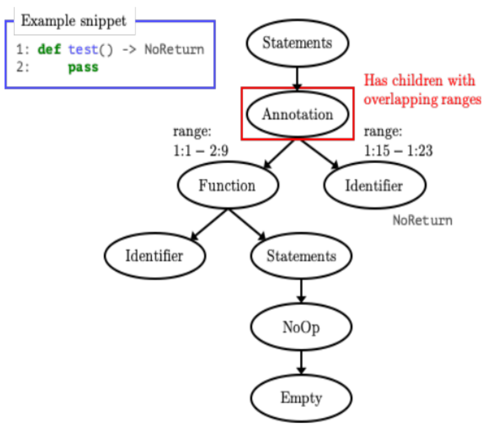
As Java is not currently supported by semantic, we employ a separate AST parser based on the javaparser project [Jav]. We use the Java Parser project contains a prebuilt java-parser-1.0-SNAPSHOT.jar. To use the code2seq Java dataset or assemble ones own Java dataset to train the CodeTransformer one can also make use of the JavaMethodExtractor-1.0.0-SNAPSHOT.jar that gathers Java methods from a folder structure of class files.This project contains a set of libraries implementing a Java 1.0 - Java 15 Parser with advanced analysis functionalities. This includes preview features to Java 13, with Java 14 preview features work-in-progress.

3.3 Obtain mapping between tokens and graph nodes

The main objective of this research is obtaining a function mapping code tokens based on ranges, i.e an adjacency representation.To do this, every token is assigned to the node in the AST with shortest source range that still encompasses the source range of the token. (*see Fig 3.1*). Making an assumption that source ranges of child nodes do not overlap would make things a lot easier when finding the smallest encompassing source range. This would imply greedily selecting at every layer in the AST the child that encompasses the token's source range, with the assumption the child at every layer would be unique or non existent. However this does not hold true for all ASTS and hence as a heuristic, we greedily select the child node with the shorter source range in case there were multiple child nodes with encompassing source ranges. This approximation seems to be sufficient in our case, and limits runtime as we do not have to consider multiple paths in the AST. It is also sufficient to stop when no child node encompasses the source range of the token, as in ASTs the source ranges of child nodes are always contained in the source ranges of their parent.

3.4 Using token mapping in denoising autoencoder

In this research we explore a novel denoising autoencoder where the noise function randomly picks a code token and masks out a given number (15-30 %) of tokens in the abstract syntax tree (AST) of the code. This is achieved in two steps: 1) Obtaining the AST by parsing the code snippets: for Java, we use the AST parser from the java-parser project, and we use Semantic for Python, JavaScript, Ruby and Go. 2) Obtaining a graph structure mapping



```
1: def test() -> NoReturn
2:     pass
```

Figure 3.1: Code snippet and corresponding AST. Source: [ZKC21]

code tokens to nodes of the AST. This is achieved by mapping every token to the node in the AST with shortest source range that still encompasses the source range of the token.

CHAPTER 4

Dataset Overview

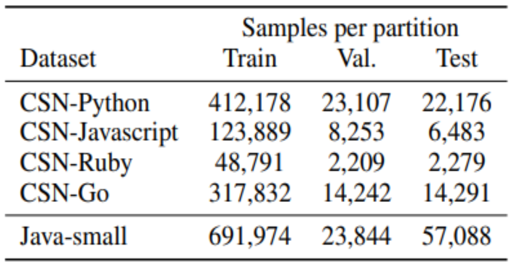
The ability to generate natural language sequences from source code snippets has a variety of applications such as code summarization, documentation, and retrieval. Sequence-to-sequence (seq2seq) models, such as PLBART have achieved state-of-the-art performance on these tasks by treating source code as a sequence of tokens. Hence having a good dataset for such research is imperative. In this research we use mainly two datasets: Code2seq (Java-small, Java-medium and Java-large) and CodeSearchNet (CSN) (Python, Ruby, Javascript, Go).

4.1 Code2seq

[ABL19] presents an alternative approach that leverages the syntactic structure of programming languages to better encode source code. Their model represents a code snippet as the set of compositional paths in its abstract syntax tree (AST) and uses attention to select the relevant paths while decoding. The code2seq dataset contains the files that hold training, test and validation sets, and a dict file for various dataset properties. In particular code methods in java-small, java-large and java-med.

4.2 CodeSearchNet (CSN)

CodeSearchNet (CSN) is a collection of datasets and benchmarks that explore the problem of code retrieval using natural language. This research is a continuation of some ideas



Dataset	Samples per partition Train	Val.	Test
CSN-Python	412,178	23,107	22,176
CSN-Javascript	123,889	8,253	6,483
CSN-Ruby	48,791	2,209	2,279
CSN-Go	317,832	14,242	14,291
Java-small	691,974	23,844	57,088

Table 4.1: CSN Dataset Statistics. Source: [ZKC21]

presented in [HWG19] and is a joint collaboration between GitHub and the Deep Program Understanding group at Microsoft Research - Cambridge. It aims to provide a platform for community research on semantic code search via the following:

- Instructions for obtaining large corpora of relevant data
- Open source code for a range of baseline models, along with pre-trained weights
- Baseline evaluation metrics and utilities
- Mechanisms to track progress on a shared community benchmark hosted by Weights & Biases

CodeSearchNet aims at engaging with the broader machine learning and NLP community regarding the relationship between source code and natural language. The dataset consistes of code snippets in the following languages: python, javascript, ruby, go in zipped jsonl files separated as train , test and valid datasets.

CHAPTER 5

Results

Some of the achievements of this research includes being able to research and identify a possible area for improving the performance of PLBART via modifying the noise function of the denoising autoencoder.

Secondly we employ techniques used in [ZKC21] and modify their code to obtain token mappings which is essentially the mapping between code tokens and AST graph nodes for different languages such as Python, Ruby, Javascript and Go. Main outcome being the zip files containing the following as output:

- tokens_batch: code tokens obtained from the batch of code snippets being processed after tokenization
- ast_graph_batch: corresponding AST graphs of the code tokens
- token_mapping_batch: mapping between code tokens and ast graph
- stripped_code_snippets: corresponding stripped code snippets
- func_names: function names of the batch
- docstrings: other strings, comments or descriptions

This token_mapping (mapping between token and graph nodes) for Python, Ruby Go can be further employed in the Denoising Autoencoder used in PLBART to modify it's noise function as discussed in Chapter 3.

Unfortunately there were issues while running this modified code for our Java Dataset. Java employs a different parser (not semantic) from the JavaParser library, and the JavaMethodExtracter provided appears to be invalid or corrupted and hence token mappings for Java code have not yet been obtained.

Due to the lack of time this script has not yet been incorporated in the PLBART which uses fairseq [OEB19] for it's PLUG tasks, and pretraining has not been run to observe corresponding performance metrics. Our hope is that since we have encoded AST structure into the self attention mechanism by obtaining the token mappings, the denoising autoencoder would mask or delete code tokens linked directly to the structure of the AST and hence the sequence-to-sequence model would have more robust learning as a consequence.

CHAPTER 6

Conclusion and Future Works

In conclusion, our objective of investigating a technique to encode Abstract Syntax Trees (ASTs) via distance based self-attention mechanism was achieved by referring to work by [ZKC21] and obtaining a mapping between code tokens and AST nodes for languages such as Python, Ruby, Javascript and Go.

Future work for this research would include:

- Obtain token mappings for the code2seq java dataset as well, after resolving issues with the invalid Java Method Extractor.
- Searching for Parsers in C, C++ and PHP and expanding to include these languages, particularly because PLBART [ACR21] supports these languages as well.
- Integrate the token mapping code with PLBART to observe (hopefully improved) results in a variety of PLUG tasks. By integration we mean modifying the denoising autoencoder which currently samples code tokens and masks or deletes these token at random. The modified noise function randomly picks a code token and masks out a given number (variable: 15-30 %) of tokens in the abstract syntax tree (AST) of the code. This uses the code token to AST mapping we have obtained in this research.

REFERENCES

[ABL19] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. “code2seq: Generating Sequences from Structured Representations of Code.” In *International Conference on Learning Representations*, 2019.

[ACR20] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. “A Transformer-based Approach for Source Code Summarization.” In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4998–5007, Online, July 2020. Association for Computational Linguistics.

[ACR21] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. “Unified Pre-training for Program Understanding and Generation.”, 2021.

[DCL19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.”, 2019.

[GGH21] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, L. Nie, and Xin Xia. “Code Structure Guided Transformer for Source Code Summarization.” *ArXiv*, **abs/2104.09340**, 2021.

[Git] Github. “github/semantic: Parsing, analyzing, and comparing source code across many languages.”.

[GRL21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. “GraphCodeBERT: Pre-training Code Representations with Data Flow.” *ArXiv*, **abs/2009.08366**, 2021.

[HWG19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. “CodeSearchNet challenge: Evaluating the state of semantic code search.” *arXiv preprint arXiv:1909.09436*, 2019.

[Jav] Javaparser. “javaparser/javaparser: Java 1-15 Parser and Abstract Syntax Tree for Java, including preview features to Java 13.”.

[LYB19] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. “Aroma: code recommendation via structural code search.” *Proceedings of the ACM on Programming Languages*, **3**:1–28, 10 2019.

[Mon17] Dominic Monn. “Denoising Autoencoders explained.”, Jul 2017.

[OEB19] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. “fairseq: A Fast, Extensible Toolkit for Sequence Modeling.” In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[PKJ21] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. "CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks.", 2021.

[VSP17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is All You Need." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, p. 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[ZKC21] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. "Language-Agnostic Representation Learning of Source Code from Structure and Context." In *International Conference on Learning Representations (ICLR)*, 2021.