CODE REPRESENTATION
LEARNING WITH PRÜFER SEQUENCES

by

Tenzin Jinpa

B. Tech, Guru Gobind Singh Indraprastha University, 2013
M. Tech, Guru Gobind Singh Indraprastha University, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The College of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

October 2021

© Tenzin Jinpa, 2021
The following individuals certify that they have read, and recommend to the College of Graduate Studies for acceptance, a thesis/dissertation entitled:

**CODE REPRESENTATION LEARNING WITH PRÜFER SEQUENCES**

submitted by **TENZIN JINPA** in partial fulfilment of the requirements of the degree of **Master of Science**

Yong Gao, Irving K. Barber Faculty of Science  
**Supervisor**

Pramjeet Gill, Irving K. Barber Faculty of Science  
**Supervisory Committee Member**

Fatemeh Hendijani Fard, Irving K. Barber Faculty of Science  
**Supervisory Committee Member**

Xiaoping Shi, Irving K. Barber Faculty of Science  
**University Examiner**

Warren Hare, Irving K. Barber Faculty of Science  
**Neutral Chair**
Abstract

An effective and efficient code representation is critical to the success of sequence-to-sequence deep neural network models for a variety of tasks in code understanding, such as code summarization and documentations, improving productivity, and reducing software development costs. Unlike the natural language, which is unstructured and noisy, programming codes are intrinsically structured, and the learning model can leverage this property of the code. A significant challenge is to find a sequence representation that captures the structural information in the program code and facilitates the training of the models.

In this study, we propose to use the Prüfer sequence of the Abstract Syntax Tree (AST) of a computer program to design a sequential representation scheme that preserves the structural information in an AST. Our representation makes it possible to develop deep-learning models in which signals carried by lexical tokens in the training examples can be exploited automatically and selectively based on their syntactic role and importance. Unlike other recently-proposed approaches, our representation is concise and lossless in terms of the structural information of the AST. To test the efficacy of Prüfer-sequence-based representation, we designed a code summarization using a sequence-to-sequence learning model on real-world benchmark datasets. The results from the empirical studies show that Prüfer-sequence-based representation is indeed highly effective and efficient, outperforming significantly all the recently-proposed deep-learning models we used as the baseline models.
Lay Summary

The application of the deep learning models in Natural Language Processing (NLP) enabled us to solve many complex NLP tasks. The success of the deep learning model in NLP inspired many researchers to apply deep learning in Software Engineering, specifically code representation learning. One key research area is how we represent a computer code used as input to train the deep learning models in code representation learning.

In this thesis, we explore the idea of lossless linear encoding of the computer code using the Prüfer-Sequence of abstract syntax tree of a code, which helps the model to learn the syntactic information of the code. We also provide a context of the code that enabled the model to learn the lexical information of the code.

To demonstrate the efficiency and the efficacy of our representation of the code, we developed a Prüfer-Sequence based learning model for code summarization (attention based Encode Decoder model with dual encoder) and compared the results with other baseline models.
# Table of Contents

Abstract ................................................................. iii

Lay Summary ............................................................ iv

Table of Contents ..................................................... v

List of Tables ......................................................... vii

List of Figures ........................................................ viii

Acknowledgements ....................................................... ix

Dedication ................................................................. x

Chapter 1: Introduction ............................................... 1

Chapter 2: Background .................................................. 4

2.1 Abstract Syntax Tree ............................................... 4
  2.1.1 Abstract Syntax Tree and Code Representation ............. 4

2.2 Linear Encoding Methods of the Code ............................ 6
  2.2.1 Flat Sequence of Tokens ..................................... 7
  2.2.2 Classical Traversal Method ................................... 8
  2.2.3 Structure Base Traversal Method ............................ 8
  2.2.4 Paths Between the Terminals Nodes ......................... 10

2.3 Prüfer Sequence of Tree and its Properties .................... 12
  2.3.1 Generation of Prüfer Sequences ............................ 13
  2.3.2 Tree Generation from the Prüfer Sequence ................ 14

2.4 Core Deep Learning Technique .................................. 15
  2.4.1 Long Short Term Memory ................................... 17
  2.4.2 Gated Recurrent Unit ....................................... 19
  2.4.3 Attention Based Encoder-Decoder Model .................. 20

2.5 Deep Learning for Code Comprehension ........................ 23
  2.5.1 Related Work .............................................. 24

Chapter 3: Prüfer Sequence Generation ............................. 26
# TABLE OF CONTENTS

3.0.1 Prüfer Sequence Generation of a Code ........................................... 26
3.0.2 Properties of the Prüfer sequence ................................................... 34

Chapter 4: Prüfer-Sequence-Based Learning Model For Code Summarization .................. 42

4.0.1 Prüfer Sequence Encoder .............................................................. 43
4.0.2 Context Encoder ......................................................................... 43
4.0.3 Attention .................................................................................... 44
4.0.4 Decoder ....................................................................................... 44

4.1 Experiment Setup .............................................................................. 45

4.1.1 Baseline .......................................................................................... 47
4.1.2 Metrics ............................................................................................ 48

4.2 Results ................................................................................................. 50

4.2.1 Performance over Source Code of Different Lengths ....................... 54
4.2.2 Flexibility of Prüfer Sequence ......................................................... 55
4.2.3 Efficiency of the Prüfer Sequence in encoding the AST of the code .... 56
4.2.4 Comment Generation Using Different AST Representation Methods ........................................ 57

Chapter 5: Conclusions .............................................................. 64

Bibliography ......................................................................................... 66

Appendices ......................................................................................... 71

Appendix A : Abstract Syntax Tree Table Generation ........................................ 71
Appendix B : Prüfer-Sequence Generation ................................................... 73
Appendix C: Context of Code Generation ...................................................... 76
List of Tables

Table 4.1  Statistics of the Java Methods . . . . . . . . . . . . . . . . . . . . . . 47
Table 4.2  Statistics of the Java comments . . . . . . . . . . . . . . . . . . . . . 47
Table 4.3  Effectiveness of Models based on Machine Translation metrics for Dataset-1 and value inside the bracket shows the performance improvements in percentage compare to the baseline models . . . . . 52
Table 4.4  Effectiveness of Models based on Machine Translation metrics For Dataset-2 and value inside the bracket shows the performance improvements in percentage compare to the baseline models . . . . . 52
Table 4.5  Flexibility of Prüfer sequence for Dataset-1 . . . . . . . . . . . . . . 56
Table 4.6  Statistics of the AST Representation Methods for Dataset-1 . . . . 56
# List of Figures

| Figure 2.1 | The Abstract Syntax Tree and Concrete Syntax Tree of the expression $7 + (2 + 3)$ | 5 |
| Figure 2.2 | The Abstract Syntax Tree of the Code | 6 |
| Figure 2.3 | The Abstract Syntax Tree of the Code | 7 |
| Figure 2.4 | SBT and DFS generated Sequence | 9 |
| Figure 2.5 | Paths Between the terminal Nodes of the AST | 11 |
| Figure 2.6 | Prüfer sequence Generation | 13 |
| Figure 2.7 | Reconstruction of Tree from Prüfer Sequence | 15 |
| Figure 2.8 | Recurrent Neural Network | 17 |
| Figure 2.9 | Long Short Term Memory | 18 |
| Figure 2.10 | Gated Recurrent Unit | 19 |
| Figure 2.11 | Encoder Decoder Model | 21 |
| Figure 3.1 | Prüfer sequence and context generation of a code | 27 |
| Figure 3.2 | The Abstract Syntax Tree of the Code | 28 |
| Figure 3.3 | Abstract Syntax Tree Table | 29 |
| Figure 3.4 | Numerically labelled Tree structure for AST in Fig.4.2 | 30 |
| Figure 3.5 | Generate Syntactic Prüfer Sequence | 31 |
| Figure 3.6 | Prüfer sequence of the AST | 37 |
| Figure 3.7 | Context Generation of the Code | 38 |
| Figure 4.1 | Prüfer Based Learning Model for Code Summarization Task | 42 |
| Figure 4.2 | Distribution of Code Length | 45 |
| Figure 4.3 | Distribution of Comment Length | 46 |
| Figure 4.4 | Frequency Distribution of Code Length | 54 |
| Figure 4.5 | BLEU score for different method lengths | 55 |
| Figure 4.6 | Time required to train Prüfer Based Learning Model for 60 epoch using different AST representation | 57 |
Acknowledgements

With great pleasure, I would like to thank The University of British Columbia Okanagan for two years of support and for providing the opportunity of teaching undergraduate students as a Teaching Assistant. My journey in a master’s program was enriching and satisfying, with opportunities to learn and expand my research understanding and technical skills.

I would like to thank Dr. Yong Gao for his constant support and mentorship for the last two years. I had benefited immensely from his deep understanding of computer science and research ideas, without which I would not be able to complete this work.

Finally, I would like to thank my committee members for their support and understanding. I am grateful for all of your advice.
Dedication

First and foremost, I would like to thank my family for their unconditional love and support, which helped me overcome all the challenges of my life and fulfill my dreams. I also like to thank Tibetan Children Village (TCV) Dharamsala and the Dalai Lama Trust for supporting my undergraduate and graduate education. I would like to thank all my friends, faculty, and other people I’ve met during my master’s program at UBC Okanagan.

I would like to dedicate this thesis to my mom, a constant source of inspiration and support throughout my life. Whatever I had achieved in my life is due to her sacrifices and hard work.
Chapter 1

Introduction

The genesis of the natural language could be traced to the early human civilization with the ability of the human species to express an idea using pictures. The modern languages developed over centuries become more structured and diverse, enabling humans to process the natural language to express complex ideas. However, the early stages of natural language processing depend on manual annotation with little or no automation limiting its applications. This changes with the evolution of modern technology, enabling us to process a massive quantity of information in a short duration. The early work on Natural Language Processing (NLP) is inspired by the work of Alan Turing [Tur50] in 1950, which demonstrates that a machine could be capable of "Thinking" through the Turing test and Hodgkin-Huxley model in 1952, which shows that the brain uses neurons in forming an electrical network. However, the progress on NLP in the following years remained stagnated, and by 1966, the National Research Council (NRC) and Automatic Language Processing Advisory Committee (ALPAC) initiated the first AI and NLP stoppage. The major paradigm shift started with developing deep learning models [TSK+20], and the availability of high computing power results in an impressive performance on various NLP tasks.

The evolution of NLP is broadly categorized into three phases, i.e., Symbolic NLP (the 1950s – early 1990s), Machine Learning NLP (1990s–2010s), and Neural NLP (present). The premise of the Symbolic NLP can be surmised by John Searle’s Chinese room experiment: Given pre-defined rules, a machine can emulate the natural language understanding by applying the rules to the data. The models based on the Symbolic NLP were constraints with pre-defined rules and domain knowledge requirements, thus failing to produce significant results. In the case of the Machine Learning models, we train the model using the large corpus of text such that during the training, the model learns the inherent association between the input and output text. Most of the classical machine learning models required hand-crafted features from the corpus as input for good performance, this constraint the models as it required domain knowledge and limited its applications to new tasks. The Neural NLP produces the most promising results in NLP tasks, it uses deep learning techniques such that it maps the input text to a low dimensional continuous feature vector rendering the requirements of hand-crafted features. The versatility of the deep learning model is validated by its applications in various tasks such as

1. Sentiment analysis

It analyzes the people’s sentiment in a text (e.g., tweets, customer reviews, and news articles) and classifies them into different categories. These categories could
be binary (positive or negative labels) or multi-class problems (with multiple labels). For example, [XMQ+19] uses the semantic information of the text as input in the recurrent neural network with the Bi-LSTM model to classify text into binary categories (positive and negative).

2. Natural Language Inference (NLI)

NLI model predicts whether the semantic meaning of one text can be inferred from another text. The model will have a parallel corpus of text such that each pair from the corpus is labeled as entailment, contradiction, and neutral. For example, [LSLW16] uses the Bi-LSTM to train the model to classify the pair of text into three categories: entailment, contradiction, and neutral, with text tokens as input.

3. Generative Question Answering

We use the deep learning models with limited success to generate natural language responses to the question based on the knowledge that the model was trained. For instance, [YJL+16] use the deep learning model were train with the parallel corpus of the question and answer such that the model will generate a natural language response to a new question it encounters.

4. Text Segmentation

A number of the text segmentation problem were resolved by applying deep learning techniques. For instance, [AA21] uses the deep learning model in the Named Entity Recognition in the non-segmented text, while [WY18] uses the deep learning model in segmenting the words in a non-segmented text.

With the advancement of the software, the size of the code became significant and the maintenance of software became expensive. Many researchers work on developing an automated tool to comprehend the code and reduced the software development cost. The use of deep learning techniques and related neural machine translation (NMT) models in code understanding has drawn much recent attention. It has been shown that methods using NMT models can achieve a much better performance than traditional Informational Retrieval (IR) techniques in tasks such as automated code summarization [HIGH16, ZWZ+20] and program property prediction [AZLY18], thus improving productivity [LVD06] and reducing software development costs [SHM+10]. Also, unlike natural languages, which are unstructured and noisy, computer programs are highly structured. It is thus critical to encode as much as possible the structural information in a sequence-to-sequence learning model and to take advantage of the encoded information in training.

In this study, we propose to use the Prüfer sequence of the Abstract Syntax Tree (AST) of a computer program to design a sequential representation scheme that preserves the structural information in an AST. Our representation makes it possible to develop deep-learning models in which signals carried by lexical tokens in the training examples can be exploited automatically and selectively based on their
syntactic role and importance. Unlike other recently-proposed approaches, our
representation is concise and lossless due to the fact that an AST can be uniquely
reconstructed from its Prüfer sequence. Empirical studies on real-world benchmark
datasets, using a sequence-to-sequence learning model we designed for code
summarization, showed that our Prüfer-sequence-based representation is indeed highly
effective and efficient, and our model outperforms significantly all the
recently-proposed deep-learning models used as the baselines in the experiments.

Chapter 2 will discuss the background knowledge, i.e., we cover a detailed discussion
on the current linear code representation methods and their limitations. We will discuss
the prüfer sequence and its properties and covered the Recurrent Neural Networks (RNN)
like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU). We also
discussed the working of the attention-based sequence-to-sequence model.

Chapter 3 presents our main idea to produce a lossless linear representation of AST
of code using the prüfer sequence. We explain the detail mechanism of generating
prüfer sequence from an AST and the context of the code from the prüfer sequence.
Furthermore, we explain the various properties of our representation compared to other
representation methods.

In chapter 4, we discuss the application of our representation. While our
representation is a programming language and task-independent, we focus its
application on code summarization in the current work to study and demonstrate its
effectiveness and efficiency in code representation learning. We discuss the results from
the series of empirical studies on widely used benchmark datasets, i.e., a) parallel
corpus of Java code and comment by Hu [HLX+20] and b) CodeXGlue dataset
(code-to-text) by Microsoft [HLX+20].
Chapter 2

Background

2.1 Abstract Syntax Tree

The Abstract Syntax Tree (AST) evolution is rooted in the compiler design to process the computer code. We use syntax-directed interpreters for simple instructions, which directly execute the code without building the intermediate representation, and for complex code from a high-level language like java or Pascal required intermediate representation. A tree data structure is used to represent the intermediate code representation for the interpreter in the compiler. The parser will generate the intermediate representation, and the interpreter uses this to interpret the code.

The Abstract Syntax Tree (AST) is an intermediate representation of the code in a tree data structure such that not every detail of the code is represented in the AST. The motivation of the AST is realized by comparing it to the Concrete Syntax Tree (CST). The CST is a syntax structure of a programming language constructed according to grammar definition. Fig. 2.1 shows the AST and CST for the instruction 7 + (2 + 3). We see that AST representation (Fig. 2.1) is more concise than the CST while preserving the essential information of the code.

We can define the Abstract Syntax Tree (AST) as the tree data structure of the code constructs according to the grammar definition of the language and provide an abstract representation (i.e., not every detail of the code is represented in the structure).

An AST will have the following properties

1. An AST can be edited or enhanced by adding information to the nodes.
2. An AST will not include all elements of the code, i.e., it skips the nonessential element of the code.
3. In the compiler design, the AST contains more information about the program as the compiler will analyze the AST in consecutive stages.

2.1.1 Abstract Syntax Tree and Code Representation

In our AST, the leaves (or terminal nodes) are labeled by tokens that contain user-defined values or reference types such as variable names, whereas internal nodes (or non-terminal nodes) are labeled by tokens that summarize the purpose of code blocks such as conditions, loops, and other flow-control statements. We note that a token labeling an internal node does not have to be from the source code or specification of the particular programming language. By slightly abusing the notion,
2.1. Abstract Syntax Tree

we call a token labeling a leave node a “lexical token” as it contains program-specific information in the source code and call a token labeling a non-terminal node a “syntactic token” as it contains generic information about the structure and the purpose of a code block. Shown in Fig.2.2 is a function in Java and its AST where the token set \{Override, Int, String, mergeErrorIntoOutput, Boolean, Commands\} are used to label the terminal nodes and the token set \{BasicType, FormalParameter, MethodInvocation, ReturnStatement, ClassCreator, ReferenceType\} are used in label the internal nodes.

In a sequence-to-sequence model for code representation learning, tokens (or their word embedding) from the source code of a computer program have to be represented as a linear sequence and are used as the input of the model. As has been shown by [HLX+18a] and [ABLY19], a linear sequence representation that makes use of structural information encoded in the AST has a great advantage over a simple flat sequence representation where tokens appear in the same order as they appear in the source code [IKCZ16]. Since AST is inherently abstract and its structure depend on the implementation, we follow the AST implementation provided by the Hu [HLX+18a] and code can be accessed by
2.2 Linear Encoding Methods of the Code

In recent years significant milestones were achieved in code representation learning using the deep learning models. One of the critical components of a deep learning model is the representation of the code (which is input into the model). To train a sequence-to-sequence model for code representation learning, we need to pass the tokens of the code in a linear sequence. We can treat the code as a flat sequence of code or use the abstract syntax tree to capture the syntactic and lexical information of the code. In the following subsections, we will discuss the four standard linear representations of the code.

Github link

1https://github.com/xing-hu/EMSE-DeepCom/blob/master/data-utils/get-ast.py
2.2. Linear Encoding Methods of the Code

Figure 2.3: The Abstract Syntax Tree of the Code

2.2.1 Flat Sequence of Tokens

Many code representation learning models with deep learning were proposed with the code tokens as input to the model. Like Iyers [IKCZ16] uses the actual code as input to the deep learning model to generate summaries to the code. For example, the java method in Fig 2.3 is represented as

```java
public int runCommand ( boolean mergeErrorIntoOutput , String commands )
    throws IOException , InterruptedException
{
    return runCommand ( mergeErrorIntoOutput , new ArrayList < String > ( Arrays . asList ( commands ) ) ) ;
}
```

This representation is straightforward, and by sending the program’s actual code as input, the model will learn the lexical information. The drawbacks of this representation are that sending the code as a flat sequence of tokens fails to capture the syntactic information of the code, and performance relies on the type of tokens used for the
2.2. Linear Encoding Methods of the Code

training and testing, as wide deviations between the two (i.e., training and testing corpus) may degrade the performance.

To overcome the limitation of the above representation, we need to represent the code in a data structure that can preserve the code’s syntactic information. We represent the code as the abstract syntax tree to preserve the syntactic information of the code and generate the linear representation of the AST (Fig. 2.3), which is input to the deep learning model.

2.2.2 Classical Traversal Method

To preserve the syntactic information of the code, we represent the code as an abstract syntax tree. To use the abstract syntax tree with the deep learning model requires a linear representation of the abstract syntax tree. We can use the classical traversal methods like Breadth-First Search (BFS) or Depth First Search (DFS). This method will generate a linear representation of the AST. For example, DFS and BFS of the AST (Fig. 2.3) is

< Annotation BasicType BasicType FormalParameter ReferenceType FormalParameter MemberReference MethodInvocation ReferenceType TypeArgument MemberReference MethodInvocation ClassCreator ReturnStatement MethodDeclaration >

< MethodDeclaration Annotation BasicType FormalParameter FormalParameter ReturnStatement BasicType ReferenceType MethodInvocation ClassCreator MemberReference ReferenceType MethodInvocation ArgumentType MemberReference ReferenceType >

The classical traversal methods generate a linear representation of the AST, which is input to the deep learning model. However, this representation of the abstract syntax tree fails to preserve the structural information of the code as such representation is lossless representation. This representation is lossy since there is no one-to-one correspondence between the tree and the sequence generated by the classical traversal methods (BFS or DFS) since we cannot reconstruct the tree from the sequence generated by the classical traversal methods. Another fundamental limitation is that sequences generated by the classical traversal methods are not unique since sequences generated by the classical traversal methods for two different codes may generate identical syntactic sequences. This result in one syntactic sequence (input to the neural network) mapping to two different comments (output to the neural network), thus confusing the neural network.

2.2.3 Structure Base Traversal Method

One of the modern methods of linear representation of the abstract syntax tree is the Structure Base Traversal (SBT) method by Hu [HLX+18a]. The steps in the SBT sequence generation are defined in Algorithm-1. Hu [HLX+18a] uses the SBT generated sequence to learn the syntactic information of the code to generate the comments for
2.2. Linear Encoding Methods of the Code

Figure 2.4: SBT and DFS generated Sequence

The SBT generated sequence of a tree uses the parenthesis to capture the sub-tree information of the code. Also, the SBT generated sequence of a tree produced similar syntactic tokens sequences as that of a DFS generated sequence if we printed all the visited nodes of the tree in the DFS traversal method (Fig. 2.4).

Algorithm-1 shows that the SBT method will generate the syntactic token each time it visited a node in the tree. For each node, we have a two-bracket, i.e., the opening bracket ", token itself n’, closing bracket "), For a tree with n nodes, the maximum length of the SBT generated sequence is of the order of 3 * n. For example, the length of the SBT generated sequence of the AST in Fig. 2.3 is 62 tokens.

```
< ( MethodDeclaration ( Annotation ) Annotation ( BasicType ) BasicType ( FormalParameter ( BasicType ) BasicType ) FormalParameter ( FormalParameter ( ReferenceType ) ReferenceType ) FormalParameter ( ReturnStatement ( MethodInvocation ( MemberReference ) MemberReference ( ClassCreator ( ReferenceType ( TypeArgument ( ReferenceType ) ReferenceType ) TypeArgument ) ReferenceType ( MethodInvocation ( MemberReference ) MemberReference ) MethodInvocation ) ClassCreator ) MethodInvocation ) ReturnStatement )
```
2.2. Linear Encoding Methods of the Code

**Algorithm 1: Structure-Based Traversal**

1. procedure SBT(r) \(\triangleright\) Traverse a tree from root
2. seq ← \(\varnothing\) \(\triangleright\) seq is the sequence of a tree after traversal
3. if !r.hasChild then
4. seq ← (r) \(\triangleright\) Add brackets for terminal nodes
5. else
6. seq ← (r \(\triangleright\) Add left bracket for non-terminal nodes
7. for c in childs do
8. seq ← seq + SBT(c)
9. seq ← seq + ) r \(\triangleright\) Add right bracket for non-terminal nodes after traversing all their children
10. return seq

The SBT generated sequence of an AST is a lossy representation as we cannot reconstruct the tree from the sequence since there is no one-to-one correspondence between the SBT generated sequence and the tree. The lossy nature of the SBT generated sequence implied that it fails to preserve the syntactic information of the code.

### 2.2.4 Paths Between the Terminal Nodes

Another modern approach of the linear representation of the AST is proposed by Alon [ABLY19], here we represent the AST as the composition of paths between the pairs of terminal nodes in the AST. In this method, a code snippet \(C\) is represented as the paths between the \(K\) pairs of terminal nodes of the AST. During decoding, the attention mechanism will choose the most relevant paths to generate the model’s output. The sequence generated from this method is a lossy encoding as we cannot reproduce a tree from a sequence of paths between the terminal nodes of the AST as there is no one-to-one correspondence between the sequence and the tree. Fig 2.5 show the paths between the four pairs of terminal nodes of AST of Java code (the four different colors of edges represent paths) from Fig 2.3

```java
< Override runCommand return runCommand mergeErrorIntoOutput , mergeErrorIntoOutput runCommand return ArrayList None String , String None
```
2.2. Linear Encoding Methods of the Code

Alon [ABLY19] representation uses the lexical tokens instead of the syntactic tokens of the AST to generate paths between the terminal nodes. The efficacy of such representation heavily relies on the identifiers in the dataset as any significant deviation of identifiers in the code may impact the model’s performance. Furthermore, another limitation of Alon [ABLY19] representation is that it is a lossy encoding of an AST of code as we cannot reconstruct the AST from the paths between the terminal nodes proposed by Alon [ABLY19]; thus model losses certain structural information of the code during training.
2.3 Prüfer Sequence of Tree and its Properties

The previous section shows that an effective and efficient linear representation of the AST needs to be a lossless representation. In this section, we will discuss a unique representation called Prüfer sequence, which is a lossless representation of a tree. We will examine the properties and the generation of the Prüfer sequence.

The Prüfer sequence is proposed by Heinz Prüfer ([Prü18, Wes00]) while solving the famous proof of Caley’s formula for trees. The proof is based on the one-to-one correspondence between the set of labeled trees and the set of such sequences. The Prüfer sequence is a unique and lossless encoding of a tree such that we can uniquely reconstruct the tree from the sequence. Formally, We can define the Prüfer sequence as

The Prüfer sequence of a tree $T$ with labelled node from 0 to $n$ is a sequence of nodes of size $n - 2$ such that each node $\{a_1, a_2, a_3, \ldots, a_{n-1}\}$ is labelled 0 to $n$ (where the repetition of the label is allowed).

Lemma 2.1. For a tree $T$ with $n$ nodes Prüfer sequence will have the length of size $n - 2$

Let $T$ be a tree with $n$ node and $m$ edges

Total number of edges in an undirected tree $m = n - 1$

Length of the Prüfer sequence $= \sum_{n \in N(T)} (\deg(n) - 1)$

$= \sum_{n \in N(T)} \deg(n) - \sum_{n \in N(T)} 1$

$= 2m - n$

$= 2(n - 1) - n$

$= n - 2$

Lemma 2.2. There are $n^{n-2}$ Prüfer sequence of length $n - 2$

By definition there are $n$ ways to choose each element of a Prüfer sequence of length $n - 2$. Since there are $n - 2$ elements to be determined, so in total, we have $n^{n-2}$ ways to choose the whole sequence.

For example let say we have Prüfer sequence of length 2 then there are $4^{4-2} = 16$ sequences of length $n - 2$ i.e. $\{(1, 1), (1, 2), (1, 3), \ldots, (4, 3), (4, 4)\}$

Lemma 2.3. For a tree $T$ with $n$ nodes, there is a one-to-one correspondence between Prüfer sequence of length $n - 2$ and the Tree $T$
2.3. Prüfer Sequence of Tree and its Properties

To prove that there exists a one to one correspondence between Prüfer sequence of length \( n - 2 \) and the Tree \( T \), we need a deterministic algorithm (Algorithm-3) that takes a Prüfer sequence and reconstruct a tree \( T \).

In the following section, we will discuss the Prüfer sequences generation steps from a tree \( T \) with labels \( \{0, 1, 2, 3, ..., n\} \) (Algorithm 2). And a deterministic algorithm (Algorithm-3) to reconstruct the tree from the Prüfer sequences.

2.3.1 Generation of Prüfer Sequences

![Prüfer sequence Generation](image)

Fig. 2.6 demonstrates the process of Prüfer sequence generation from a tree labeled from 0 to 7 node. Initially the Prüfer sequence (represented by \( P \)) is empty. Remove the terminal node 1 and include the node 0 in Prüfer sequence so \( P = \{0\} \). Now remove the terminal node 4 and add the node 2 to Prüfer sequence so \( P = \{0, 2\} \). Next remove the terminal node 2 and add 0 to Prüfer sequence so \( P = \{0, 2, 0\} \). Now remove node
2.3. Prüfer Sequence of Tree and its Properties

Algorithm 2: Prüfer sequence generation from tree T

Let us consider a Tree T with N nodes such that tree is labeled as \{0, 1, 2, ..., N\}

Step 1 Removed the smallest terminal node \(x\) from the tree \(T\)

Step 2 Add node \(y\) connected to \(x\) as the next node in the prüfer sequence

Step 3 Repeat Step 2 and 3 until only two nodes is left in the tree \(T\)

0 and add 3 to Prüfer sequence so \(P = \{0 \ 2 \ 0 \ 3\}\). Next node 3 is removed and node 5 is added to Prüfer sequence so \(P = \{0 \ 2 \ 0 \ 3 \ 5\}\). Now remove the terminal node 6 and add 5 to Prüfer sequence so \(P = \{0 \ 2 \ 0 \ 3 \ 5 \ 5\}\). We stop the process as tree has only two nodes and the final Prüfer sequence is

\[ P = \{0 \ 2 \ 0 \ 3 \ 5 \ 5\} \]

2.3.2 Tree Generation from the Prüfer Sequence

Prüfer sequence inherently possesses a unique property that it is a lossless encoding of the tree. This implied that the Prüfer sequence preserved the structural information of the tree such that we can reconstruct the tree from the given Prüfer sequence. Algorithm-3 show the steps in reproducing a tree from a given Prüfer sequence \(S\) such that each element in the Prüfer sequence can take 1, 2, 3, ..., \(n\) labels.

For example Fig. 2.7 demonstrate the tree generation from the initial Prüfer sequence \(P = \{0 \ 2 \ 0 \ 3 \ 5 \}\). Here \(P\) represent the Prüfer sequence and \(L\) is the set of all possible sequence labels.

Find the smallest node in \(L\), i.e., 1 such that 1 is not present in the Prüfer sequence \(P\). Remove the node 1 and leading node 0 in Prüfer sequence from the set \(L\) and sequence \(P\) respectively. Join the node 1 and 0. Now remove the node 4 (result from the find function in set \(L\)) from the set \(L\) and node 2 (leading node in the sequence) from sequence \(P\). Join node 4 and 2. Next, remove the node 2 (result from the find function in set \(L\)) from the set \(L\) and node 0 (leading node in the sequence) from sequence \(P\). Join the node 0 and 2. Next, remove the node 0 (result from the find function in set \(L\)) from the set \(L\) and node 3 (leading node in the sequence) from sequence \(P\). Join the node 0 and 3. Next, remove the node 3 (result from the find function in set \(L\)) from the set \(L\) and node 5 (leading node in the sequence) from sequence \(P\). Join node 3 and 5. Next, remove the node 6 (result from the find function in set \(L\)) from the set \(L\) and node 5 (leading node in the sequence) from sequence \(P\). Join the node 6 and 5. At this stage, we are left with two nodes in \(L\), i.e., node 5 and node 6 and no nodes in Prüfer sequence \(P\). Remove the nodes 5 and 6 from \(L\) and join the node 5 and 6.
2.4 Core Deep Learning Technique

A Recurrent Neural Network (RNN) is a class of neural networks where the output of the previous node is input to the next node and performs well for tasks having sequential information. For example, RNN models design weather forecasting (time-series data) and text sentiment classification (sequence of natural language words). The RNN is called “recurrent” as every node in the RNN performs the same task with output depends on the previous computations. In principle, an RNN model can have any length of input and can preserve the long-term dependencies. For example, Fig. 2.8 shows the vanilla architecture of an RNN model such that $x_1$ to $x_n$ represents the model’s input. And $U, V$, and $W$ are the weight of the model. For each node $x_t$ at time step $t$ in RNN, multiply the input $x_t$ with weight $w_x$ and hidden state $h_{t-1}$ with $w_h$ and take the sum of the products and add a bias (i.e. $b$). Apply the sigmoid function on the sum, and output is the node’s current hidden state $h_t$. To generate the output $y_t$, take the product of the
2.4. Core Deep Learning Technique

**Algorithm 3:** Reconstruction of Tree from the pr"ufer sequence

Let us consider a pr"ufer sequence \( S \) of length \( n-2 \) and such that each element \( a_i \) in the pr"ufer sequence can take any label from \( \{1, 2, 3, \ldots, n\} \)

\[
P = \{a_1, a_2, a_3, \ldots, a_n\}
\]
\[
L = \{1, 2, 3, \ldots, n\}
\]

Step 1 Find the smallest value \( x \) in set \( L \) such that \( x \) is not in Sequence \( P \)

\[
\text{Find}(x \in L : x \notin S)
\]

Step 2 Join the node label \( x \) and leading node \( a_i \) in the sequence \( P \)

Step 3 Delete node \( x \) in \( L \) and leading node \( a_i \) in sequence \( P \)

\[
P = \{a_1, a_2, a_3, \ldots, a_n\} - a_i
\]
\[
L = \{1, 2, 3, \ldots, n\} - x
\]

Step 4 Repeat the step 1 to 3 until only two elements is left in the set \( L \)

Step 5 Join the last two element

\( h_t \) and \( V \) and add bias \( c \). Apply the output function like the softmax function.

\[
h_t = \sigma(U x_i + W h_{t-1} + b)
\]
\[
y_t = O(V h_t + c)
\]

where for each each input \( x \), we defined a sigmoid function as

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]

such that the output of the sigmoid function is between 0 and 1. The output function \( O \) is a softmax function and we defined the softmax function as

\[
O(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}
\]

where softmax function map a vector of real values to the probability vector of same dimension such that the sum of the probability vector value is 1.

We use backward propagation to update the weights of the network. In general, RNN can preserve the long-term dependencies to have any length input series, but in practice,
2.4. Core Deep Learning Technique

Figure 2.8: Recurrent Neural Network

it loses its long-term dependencies due to the vanishing or exploding gradient problem. To resolve the vanishing gradient problem in the RNN models and thus preserve the long-term dependencies, many variants of the RNN were proposed, like Long Short Term Memory (LSTM) by [HS97] or Gated Recurrent Unit (GRU) by [CGCB14].

2.4.1 Long Short Term Memory

In theory, any Recurrent Neural Network (RNN) can preserve long-term dependencies. However, in practice, due to the vanishing or exploding gradient problem, an RNN cannot preserve the long-term dependencies. This conundrum was resolved by [HS97] proposing a new model called LSTM (Long Short Term Memory), which prevents the vanishing gradient, thus preserving the long-term dependencies. The central idea of an LSTM is the memory cell, which preserves the states for a long time using the number of gates, i.e., Forget gate, Input gate, and Output gate.

Forget Gate

This gate helps us to identify the values in the cell state $C_{t-1}$, which could be forgotten. This is achieved by passing the input value $x_t$ and hidden state $h_{t-1}$ to a sigmoid function, which generates value from 0 to 1 for each of the cell state $C_{t-1}$. For instance, any value in the cell state whose sigmoid value is 0 could be forgotten.
2.4. Core Deep Learning Technique

\[ f_t = \sigma(W_f \times [h_{t-1}, x_t] + b_f) \]

**Input Gate**

The input gate helps to decide what new information needs to be kept in the cell state. For this, the input is passed through the sigmoid layer, which decides what value to be updated, and then the tanh layer (it is also like logistic sigmoid with range of the tanh function between -1 to 1) creates a vector of new candidate values, \( \hat{C}_t \). Finally, the previous cell state \( C_t \) is updated using \( \hat{C}_t \).

\[
\begin{align*}
   i_t &= \sigma(W_i \times [h_{t-1}, x_t] + b_i) \\
   \hat{C}_t &= \tanh(W_c \times [h_{t-1}, x_t] + b_c) \\
   C_t &= f_t \times C_{t-1} + i_t \times \hat{C}_t 
\end{align*}
\]

**Output Gate**

This gate decides what information needs to go out of the cell. Initially, a sigmoid layer will decide what information needs to go out, and then a new updated cell state \( C_t \) passes through the tanh function multiplied by the sigmoid layer’s output, ensuring that only those value that we have decided goes to the output.

\[
\begin{align*}
   o_t &= \sigma(W_o \times [h_{t-1}, x_t] + b_o) \\
   h_t &= \tanh(C_t) \times o_t 
\end{align*}
\]
2.4. Core Deep Learning Technique

\[ O_t = \sigma(W_O \ast [h_{t-1}, x_t] + b_o) \]
\[ h_t = O_t \ast \tanh(C_t) \]

2.4.2 Gated Recurrent Unit

In this subsection, we will discuss another recurrent neural network called Gated Recurrent Unit (GRU) proposed by [CGCB14] that helps prevent vanishing gradient or exploding gradient problems in RNN models. The GRU has a structural similarity with that of the LSTM as it uses gates to avoid the gradient problem, i.e., Update Gate and Reset Gate deciding what information should be passed to the output. The GRU is computationally less expensive than the LSTM as it requires fewer gates than the LSTM.

**Update Gate**

The update gate helps the model decide how much information from the past (i.e., from previous time steps) needs to be passed to the future. we calculate the update gate \( z_t \) at time step \( t \) as

\[ z_t = \sigma(W_z x_t + U_z h_{t-1}) \]
where $x_t$ is the input to the network unit at time step $t$ and is multiplied by weight $W_z$ and the $h_{t-1}$ is the hidden state (from the previous state) and is multiplied by the weight $U_z$. We summed the two products and applied a sigmoid function, ensuring that the output of the $z_t$ is between 0 and 1.

**Reset Gate**

The reset gate will decide how much information from the past needs to be forgotten. The functionality of the reset gate is similar to that of the forget gate of the LSTM. For an input $x_t$ at timestep $t$, $r_t$ is computed as

$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

The formula for the forget gate is identical to that of the update gate as we applied the sigmoid function on the summed of the product of $x_t$ and hidden state $h_{t-1}$ with their respective weights. The difference between the update gate and the reset gate is its weight and usage.

The reset gate is used to generate a temporary memory content $h'_t$, which holds the relevant information from the past. We calculate the $h'_t$ as

$$h'_t = \tanh(W_h x_t + U_h (r_t \odot h_{t-1}))$$

multiply the input $x_t$ with $W_h$ and element-wise product of $r_t$ and $h_{t-1}$, which determine what needs to be removed from the previous time step, with $U_h$. Take the sum of the product and apply a nonlinear activation function tanh on it.

Update gate help to calculate the $h_t$, which holds the current unit $t$ information, and this information is passed to the next unit in RNN. Update gate decides what information from the previous state $h_{t-1}$ and $h'_t$ need to be kept. The $h_t$ is calculated as

$$h_t = z_t \odot h'_t + ((1 - z_t) \odot h_{t-1})$$

We apply the element-wise multiplication of $(1 - z_t)$ and $h_{t-1}$ and $(z_t)$ and $h'_t$. After the multiplication, we add the results. Thus ensuring that we only get the relevant information, for instance, let say if $z_t$ is close to 1, then $1 - z_t$ will be close to zero, ensuring that most information came from the $z_t$.

**2.4.3 Attention Based Encoder-Decoder Model**

The traditional Deep Neural Networks (DNN) are powerful models used in speech and image recognition with excellent performance. However, such a model required large labeled datasets to encode inputs and outputs with vectors of fixed dimensionality. These models cannot be used to map sequences to sequences as we would encode our input to a fixed-length vector using a technique like Word2Vec. In such encoding, the order of the words is not preserved; so when we feed our vector to model, it fails to recognize the order of the words, thus losing important information about the input. This problem was resolved by [SVL14], which proposed an encoder-decoder sequence-to-sequence model.
In the following subsection, we describe the working of the encoder-decoder model and its limitations, which encourages the development of an attention based encoder-decoder model.

An encoder-decoder model at a high level can be summarised with two blocks, i.e., an encoder block and a decoder block connected by a fixed-length context vector, which represents a summary of the input sequence.

**Encoder**

An encoder is responsible for processing each input token such that it encodes the entire input sequence into a fixed-length context vector. The final hidden state of the encoder encapsulates (in theory) all the input information and feeds this as input to the decoder. The encoder part could be an LSTM or GRU, and the output at each timestep of the encoder is discarded. Formally for an input sequence \((x_1 \ldots x_n)\), the encoder reads the input sequence and encode them, here the \(h_t\) is the hidden state of the encoder at time step \(t\) and \(c\) is the context vector. Here RNN could be a LSTM or a GRU while \(c\) is a vector generated from the sequence of the hidden states and \(q\) is a nonlinear function.

\[
h_t = RNN(h_{t-1}, x_t) \\
c = q(h_1, \ldots, h_n)
\]
2.4. Core Deep Learning Technique

Decoder

The final hidden state of the encoder is passed to the decoder as its initial input state called the context vector. With the help of the context vector $c$ and all the previous predicted words $<y_1, ..., y_{t-1}>$, the decoder tries to predict the model’s output $y_t$ at that timestep $t$.

$$p(y) = \prod_{t=1}^{n} p(y_t | y_1, ..., y_{t-1}, c)$$

where $y = (y_1, ..., y_t)$ and each conditional probability is modeled as

$$p(y_t | y_1, ..., y_{t-1}, c) = g(y_{t-1}, s_t, c)$$

where $g$ is a nonlinear function that compute the $y_t$ and $s_t$ is the hidden state of the decoder.

Unlike the encoder, the decoder training and the testing have different working. We add $<\text{start}>$ and $<\text{end}>$ tokens at the beginning and the end of the input fed to the decoder, indicating the start and the end of the sentence. During the training, we feed the actual output $y_{t-1}$ tokens from the previous time-step as input to the model at time-step $t$ (feeding the actual input instead to predicted is called the “Teacher Forcing”). The output of the decoder at time-step $t$ ($y_t'$) is the highest probability distribution over the entire vocabulary in the output dataset computed by the Softmax activation function. Finally, losses are calculated for each time step using the categorical cross-entropy loss function between the predicted ($y_t'$) and actual output ($y_t$). The errors are then backpropagated through time to update the parameters of the model. During the testing, we will feed the predicted output from the previous time-step $t-1$ to the current time step $t$ to generate the current output prediction while the rest of the working remains the same.

Attention

In an Encoder-Decoder model, final hidden state (which encoded all the input sequence information) will act as an initial input to the decoder. However, encoding all the input to a single vector has some drawbacks. Firstly, when an encoder is forced to encode all the information into a single vector, it may forget some of the encoded information. Secondly, during the decoding, the decoder will treat every encoded information with equal importance and focus on the entire context vector $c$ during the decoding at time step $t$, which is inefficient as only a subset of encoded information is relevant at time step $t$ of decoding.

The attention model by [BCB15] proposed that at each decoder step, the decoder will decide which part of the input is more relevant. This is determined by designing
a decoder that learns to align and translate jointly. Each time the model generates an output word, it searches for a set of positions in source sentences where the most relevant information is concentrated. The model then predicts output based on the context vector associated with the source positions and the previously generated outputs. We define the conditional probability for decoder as:

\[ p(y_t|y_1, \ldots, y_{t-1}, X) = g(y_{t-1}, s_t, c_t) \]

where \( s_t \) is the hidden state at time-step \( t \) and computed as:

\[ s_t = f(s_{t-1}, y_{t-1}, c_t) \]

here the probability is conditioned over distinct context vector \( c_t \) for each target word \( y_t \). The context vector \( c_t \) is the weighted sum of the annotation of \( h_t \) (encoded by the encoder) and we compute the context \( c_t \) as

\[ c(t) = \sum_{k=1}^{m} \alpha_{kt} h_k \]

The weight \( \alpha_{kt} \) of each annotation \( h_k \) is computed by a softmax function as

\[ \alpha_{kt} = \frac{exp(e_{kt})}{\sum_{l=1}^{n} exp(e_{lt})} \]

where

\[ e_{kt} = a(s_{t-1}, h_k) \]

where \( a \) is the alignment model which scores how much a input around position \( k \) and the output at position \( t \) match. This alignment model depend on the decoder hidden state \( s_{t-1} \) and \( k-th \) annotation \( h_k \) of the input sequence.

2.5 Deep Learning for Code Comprehension

In the past few decades, rapid technological growth resulted in significant code documentation and the high cost of its maintenance. For example, according to study [XBL+18], developers spend around 59% of their time in program comprehension in a software development life cycle. This results in the need to automate the program comprehension to reduce the software development cost and increase productivity.
Early approaches to program comprehension are grouped into manually crafted code generation or information retrieval (IR). For instance, Moreno [MAS+13] uses the heuristic and stereotyped to select the information which will be included in the comments generated for the Java class. Haiduc [HAM10] uses the IR approaches to generate summaries for classes and methods. Though promising, IR approaches such as Vector Space Model (VSM) and Latent Semantic Indexing (LSI) have two significant limitations: firstly, when the identifiers and methods are poorly named, they fail to extract accurate keywords to identify the code snippet. Secondly, they rely on the degree of the similarity of code snippets and whether such code snippets can be retrieved.

The recent year saw much attention in applying deep learning techniques and related neural machine translation (NMT) models in code representation learning. For example the use of NMT models in tasks such as code summarization [HIGH16, LJM19, ZWZ+20] and program property prediction [AZLY18] provide better performance than the traditional Information Retrieval (IR) techniques thus improving productivity [KMCA06, LVD06] and reducing software development costs [SHM+10, XBL+18]. Unlike natural languages like English, which are highly unstructured and noisy, program codes are highly structured. To take advantage of this intrinsic property of the code and enhance the model’s performance, it is critical that we encode as much as possible the structural information of the code in the sequence-to-sequence learning model.

Several methods have been proposed to incorporate the structural information in the Abstract Syntax Tree (AST) of a computer program in a sequence-to-sequence model. For example, the structure Based Traversal (SBT) method proposed by Hu [HLX+18a], where the sequence is generated by depth-first search traversal of the AST and augmented by the parentheses pairs to retain the sub-tree information. Also, Alon [AZLY18] proposed the used of the paths between pairs of terminal nodes in the AST to represent the program code. While these methods are effective compared to the most straightforward representation, the choices of the traversal method and the ordering of the tokens appear to be arbitrary.

## 2.5.1 Related Work

The use of deep learning models to learn a code representation has attracted much recent attention. Iyer [IKCZ16] showed that the attention-based sequence to sequence model provides better results in code representation learning than the classical Information Retrieval (IR) methods. Many successful deep learning models were developed in recent years for various tasks such as code summarization [WZY+18, HLWM20, ACRC20], code generation [BGB+17], and code retrieval [ATGW15].

Several approaches have been investigated to making use of syntactic and structural information, explicitly or implicitly, in representation learning from the source code of computer programs. Rayche [RBVK16] used the relations in the abstract syntax tree as a feature for training the model, and Bielik [BRV16], and Raychev [RBV16] used the
paths in an AST for identifying the context node. Hu [HLX+18a] proposed a Structure-Based Traversal (SBT) method to represent the AST for the code summarization task. Hu [HLX+20] used the tokens of code and the SBT generated sequence as an input to the model. SBT captures the code’s structural information, while the code tokens will help in learning the code’s lexical information. On close inspection of the SBT sequence, we can observe the structural similarity between the SBT and depth-first search traversal method. Alon [ABLY19] used the paths between the AST’s terminal nodes for code summarization and code captioning. The main idea was to represent the code snippet C with paths between k pairs of terminal nodes. A new set of paths are sampled during every training iteration to avoid bias. In addition to these efforts of directly using the structural information in the ASTs in a sequence-to-sequence model, [ACRC20] explored the possibility of exploiting structural information implicitly with a transformer model enhanced by pairwise semantic relationships of tokens in the model’s attention mechanism. A significant downside of a transformer-based approach is the increase in model complexity (quadratic in the code length) and thus the sample complexity and the computational complexity.

Another approach in code representation learning is the tree-based or graph-based learning methods of the source code. For example, Tree-Based Convolutional Neural Network (TBCNN) by [ZWZ+19] is a convolutional-based supervised learning model such as source code classification using the AST of the code. It uses the bottom-up (i.e., from leaf to root ) encoding of the tree such that it incorporates some global information and improves its localness. Similarly, Tree-based Long Short-Term Memory models (Tree-LSTM) like Child-Sum Tree-LSTM [TSM15] uses the recursive encoding of the current inputs with its children states to update the state across the tree. Alexander [LHWM20] used the graph neural network on AST for the code summarization task. While the use of these general-purpose graph neural networks in code understanding has its merit, we believe that using state-of-the-art sequence-to-sequence models enhanced by domain-specific structural information provides the best trade-offs and opportunities.
Chapter 3

Prüfer Sequence Generation

The previous discussion shows that the Abstract Syntax Tree representation of a code plays a vital role in learning the code representation. In chapter 2, we discussed various methods like classical traversal methods such as Breadth-First Search (BFS), and Depth First Search (DFS) and modern methods like Structure Base Traversal (SBT) by Hu [HLX+18a] and paths between the terminal nodes of the abstract syntax tree by Alon [ABLY19]. We also discussed the unique representation of a tree called Prüfer sequence, which is a lossless encoding of the tree as there is a one-to-one correspondence between the Prüfer sequence and the tree. Thence, we can apply the Prüfer sequence to represent the abstract syntax tree of the program code. We also use the Prüfer sequence to construct a context of the code that helps in learning the lexical information of the code.

Algorithm 4: Pseudocode for Prüfer Sequence Generation from a Abstract Syntax Tree of code

Step 1 Given an AST of the code that is labeled by program tokens

Step 2 Use a fixed mapping to map each token in the given token set to a unique integer and

Step 3 Use it as the integer label of the AST-node that is labeled by the token to generate integer-labeled AST

Step 4 Construct Prüfer sequence from this integer-labeled AST

Step 5 Mapped the Prüfer sequence in step 4 to sequence of syntactic tokens, which we call the “Syntactic Prüfer Sequence”

3.0.1 Prüfer Sequence Generation of a Code

In this subsection, we discuss the mechanism for generating the Prüfer sequence of the program code. Fig 3.1 shows the overall architecture of the Prüfer sequence and context generation of a java code and algorithm-4 provide the Pseudocode for Prüfer sequence generation from an AST of code.
Chapter 3. Prüfer Sequence Generation

Step 1. Given a Computer Code Generate the AST

As discussed in chapter 2, given a code, we generate an AST such that the internal nodes of the AST hold the syntactic information of the code. And the terminal nodes of the AST hold the user-defined value of the AST like identifiers. For example, Fig. 3.2 shows a java method and corresponding AST, such that internal nodes are `<BasicType FormalParameter MethodInvocation ReturnStatement ClassCreator ReferenceType>` and the terminal nodes are `<Override Int String mergeErrorIntoOutput Boolean Commands>`
Step 2. Map each token in the given token set to a unique integer

We use a fixed mapping to map each token in the given token set to a unique integer and use it as the integer label of the AST-node that is labeled by the token. We hold the mapping information of an AST in an AST-Table with four attributes, i.e., \(< parent, children, type, value >\). The parent and children are the integers that hold the edge information of the AST and “type” shows the syntactic type of the node, for example, IfStatement, Block, WhileStatement, and “value” shows the lexical token occurring in the code, for example, identifiers like the variable name or the method name of the code. Fig 3.3 shows an instance of the AST-Table.
Step 3. Generating Syntactic Prüfer Sequence of the Code

Given an integer-labeled AST of the code (Fig. 3.4), we generate the Prüfer sequence of the tree using Algorithm-2 (Chapter 2), i.e., Given a tree $T$ with $n$ nodes labeled by the integers $\{1, \ldots, n\}$, its Prüfer sequence is a sequence of $(n - 2)$ node labels (i.e., integers) and can be formed by successively removing the leaf with the smallest label and including the label of its parent as the next node label in the Prüfer sequence. The process stopped when only two nodes were left in the tree. The prüfer sequence of the tree in Fig. 3.4 is

$$\{0 \ 0 \ 3 \ 0 \ 5 \ 0 \ 7 \ 8 \ 7 \ 10 \ 12 \ 11 \ 10 \ 14 \}$$
Step 5. Generate Syntactic Prüfer Sequence

The Prüfer sequence constructed from this integer-labeled AST is then mapped back to a sequence of syntactic tokens (Fig.3.5), which we call the “syntactic Prüfer sequence” and is used as part of the input sequence to our learning model. The syntactic Prüfer sequence for the Java method in Fig. 3.1 is

\{MethodDeclaration, MethodDeclaration, FormalParameter, MethodDeclaration, FormalParameter, MethodDeclaration, ReturnStatement, MethodInvocation, MethodInvocation, ClassCreator, TypeArgument, ReferenceType, ClassCreator, MethodInvocation\}

Note that in the above sequence, a syntactic token may appear multiple times and at different positions. Also, note that the terminal nodes never appear in the sequence. The significance and relevance of such properties of the syntactic Prüfer sequence will be discussed below and further explored in the next chapter on the design of our learning model and our empirical studies.
Chapter 3. Prüfer Sequence Generation

Figure 3.5: Generate Syntactic Prüfer Sequence

Following are the examples of Prüfer sequence for java codes

1. Prüfer-Sequence : MethodDeclaration MethodDeclaration FormalParameter MethodDeclaration FormalParameter MethodDeclaration BinaryOperation BinaryOperation IfStatement ClassCreator ClassCreator ThrowStatement BlockStatement IfStatement MethodDeclaration ReturnStatement

```
public static < T > T requireNonNull ( T object, String message )
{
    if ( object == null )
    {
        throw new NullPointerException(message);
    }
    return object;
}
```

2. Prüfer-Sequence : MethodDeclaration TypeArgument ReferenceType MethodDeclaration FormalParameter MethodDeclaration ReturnStatement

```
public static < T > Callable < T > justCallable ( T value )
{
    return new JustValue< Object , T >( value );
}
```

3. Prüfer-Sequence : MethodDeclaration TypeArgument ReferenceType MethodDeclaration FormalParameter MethodDeclaration ReturnStatement

```
public static < T > Callable < T > justCallable ( T value )
{
    return new JustValue< Object , T >( value );
}
```
Chapter 3. Prüfer Sequence Generation

public Throwable blockinggetError(long timeout, TimeUnit unit)
{
    if(getCount() != NUM_)
    {
        try{
            BlockingHelper.verifyNonBlocking();
            if(!await(timeout,unit))
            {
                dispose();
                throw ExceptionHelper.wrapOrThrow(new TimeoutException( timeoutMessage(timeout,unit)));
            }
        }
        catch ( InterruptedException ex )
        {
            dispose();
            throw ExceptionHelper.wrapOrThrow(ex);}
    }
    return error;
}

3. Prüfer-Sequence : MethodDeclaration FormalParameter MethodDeclaration
    FormalParameter MethodDeclaration BinaryOperation BinaryOperation
    IfStatement StatementExpression TryStatement MethodInvocation
    MethodInvocation IfStatement StatementExpression BlockStatement
    MethodInvocation ClassCreator MethodInvocation MethodInvocation
    MethodInvocation TryStatement BlockStatement MethodInvocation
    CatchClause StatementExpression StatementExpression BlockStatement
    MethodInvocation TryStatement MethodInvocation MethodInvocation
    MethodInvocation ThrowStatement BlockStatement MethodInvocation
    MethodDeclaration ReturnStatement

public Throwable getError()
{
    Object o = value;
    if(NotificationLite.isError(o))
    {
        return NotificationLite.getError(o);
    }
    return null;
}
4. Prüfer-Sequence : MethodDeclaration MethodDeclaration VariableDeclarator
LocalVariableDeclaration LocalVariableDeclaration MethodDeclaration
MethodInvocation IfStatement MethodInvocation ReturnStatement
BlockStatement IfStatement MethodDeclaration ReturnStatement

```java
public void startUnbounded()
{
    if (SubscriptionHelper.setOnce( upstream, EmptySubscription.INSTANCE ))
    {
        queue = new SpscLinkedArrayQueue<T>(bufferSize);
    }
}
```

5. Prüfer-Sequence : IfStatement MethodInvocation MethodInvocation
IfStatement BlockStatement StatementExpression Assignment Assignment
ClassCreator TypeArgument ReferenceType ClassCreator MethodDeclaration

```java
public long calculateDelay(TimeUnit unit)
{
    float delta = variancePercent/NUM_;
    float lowerBound = NUM_ - delta;
    float upperBound = NUM_ + delta;
    float bound = upperBound - lowerBound;
    float delayPercent = lowerBound + ( random.nextFloat().bound );
    long callDelayMs = ( long ) ( delayMs.delayPercent );
    return MILLISECONDS.convert(callDelayMs,unit);
}
```

6. Prüfer-Sequence : MethodDeclaration FormalParameter MethodDeclaration
LocalVariableDeclaration BinaryOperation BinaryOperation VariableDeclarator
LocalVariableDeclaration MethodDeclaration LocalVariableDeclaration
BinaryOperation BinaryOperation VariableDeclarator LocalVariableDeclaration
MethodDeclaration LocalVariableDeclaration BinaryOperation BinaryOperation
VariableDeclarator LocalVariableDeclaration MethodDeclaration
LocalVariableDeclaration BinaryOperation BinaryOperation VariableDeclarator
LocalVariableDeclaration MethodDeclaration LocalVariableDeclaration
BinaryOperation BinaryOperation VariableDeclarator LocalVariableDeclaration
MethodDeclaration
Chapter 3. Prüfer Sequence Generation

```java
public OAuth2ClientConfigurer<HttpSecurity> oauth2Client() throws Exception {
    OAuth2ClientConfigurer<HttpSecurity> configurer = getOrApply(new OAuth2ClientConfigurer<>());
    this.postProcess(configurer);
    return configurer;
}
```

7. **Prüfer-Sequence**: TypeArgument ReferenceType MethodDeclaration
   TypeArgument ReferenceType LocalVariableDeclaration ClassCreator
   MethodInvocation VariableDeclarator LocalVariableDeclaration
   MethodDeclaration MethodInvocation This StatementExpression
   MethodDeclaration ReturnStatement

3.0.2 Properties of the Prüfer sequence

The syntactic Prüfer sequence can be regarded as a “transformed” and “quantified” version of an AST and the corresponding source code where

1. the frequency with which a syntax token appears is decided by the degree of the corresponding AST node it labels and quantifies the “importance” of the token (measured the size of the code block it controls);

2. the positions of the appearances of syntactic tokens in the Prüfer sequence are decided by the position of the corresponding node in the tree; and

3. a lexical token labeling a leave node of an AST never appears in the Prüfer sequence, but its “significance” can be measured by the syntactic importance of the parent of the leave node.

This is in sharp contrast with all other recently proposed sequential representations of a source code and its AST, where all tokens are treated equally, and their positions partially capture their roles in the AST.

Supported by observations from our empirical studies (discussed in the next chapter), we believe that these are the properties that make it possible (or much easier) for our learning models to exploit information in the training examples that are hard, if not impossible, for other recently-proposed learning models to detect.

As we observed in our experiments (Chapter 4), other properties that distinguish our Prüfer-sequence representation from exiting representations and play important roles in the performance of our learning model are as follows.
1. **Uniqueness and Lossless Representation**

   To our best knowledge, all existing sequential representations of source code and its AST [AZLY18, HLX+18a] are lossy encoding in the sense that the original AST cannot be uniquely reconstructed. Our Prüfer-sequence representation is a lossless encoding because, given a fixed syntactic-token-to-integer mapping, there is a one-to-one correspondence between the set of ASTs and their syntactic Prüfer sequences. For example, in chapter 2, we show the lossless encoding of the prüfer sequences using the lemma 2.3, as we can reconstruct the original tree from the prüfer sequences. We note that none of the previous representations, such as classical traversal methods like Breadth-First Search or Depth-First search or more modern techniques like Structure Base Traversal [HLX+18a] or paths between the terminals [ABLY19] has such unique properties. The lossless representation of the AST ensures that the prüfer sequences preserved syntactic information of the code, and this property may help improve the ability of a learning method to distinguish or detect subtle differences in training examples.

2. **Shorter Input Length (or Lower Dimension)**

   For an AST of length of \( n \) nodes, the length of our representation is in the worst case is \( (n - 2) + (n - 1) = 2n - 3 \). In comparison, the length of representation proposed by Hu [HLX+18a] is \( 3n \) while the length of the representation proposed by Alon [IKCZ16] is in \( \Omega(n^3) \) in the worst case. A shorter representation of an AST required a lower dimension of the model, which helped in faster training of the model and reduced the long-term dependencies in the sequence. For example, let us consider an AST of a java method with 17 nodes (shown in Fig. 3.2), the length of each representation is

   (a) For our representation, the length of the AST is 28 tokens i.e,
   ```plaintext
   {MethodDeclaration MethodDeclaration FormalParameter MethodDeclaration FormalParameter MethodDeclaration ReturnStatement MethodInvocation MethodInvocation ClassCreator TypeArgument ReferenceType ClassCreator MethodInvocation Annotation Override BasicType int BasicType boolean ReferenceType String MemberReference mergeErrorIntoOutput ReferenceType String MemberReference commands }
   ```

   (b) For structure based traversal (SBT) method by Hu [HLX+18a], the length of the AST representation is 53 tokens i.e,
   ```plaintext
   { ( MethodDeclaration ( Annotation ) Annotation ( BasicType ) BasicType ( FormalParameter ( BasicType ) BasicType ) BasicType ) FormalParameter ( FormalParameter ( ReferenceType ) ReferenceType ) FormalParameter ( ReturnStatement ( MethodInvocation ( MemberReference ) MemberReference ( ClassCreator ( ReferenceType ) TypeArgument ) commands )
   ```

   35
(c) For Alon [IKCZ16], the length of the AST when \( K \) is set to 5 is 28 nodes

\[
\text{Override runCommand return runCommand mergeErrorIntoOutput , mergeErrorIntoOutput runCommand return ArrayList None String , String None ArrayList return Arrays.asList commands , mergeErrorIntoOutput runCommand return ArrayList Arrays.asList commands , boolean mergeErrorIntoOutput runCommand commands String }
\]

In our experiment, we observed that the Prüfer sequence representation has an average length of 100.81 tokens while the representation based on SBT [HLX+18a] has the 193.71 tokens to represent the same AST corpus(Table 4.6). This results in the faster training of the model, and we will discuss the empirical results in chapter 4.

3. Natural Separation of Terminal and Non-Terminal Nodes

As we pointed noted in chapter 2 of Prüfer sequence generation that the terminal node of a tree does not appear in the Prüfer sequence. This provides a natural way to separate the terminal nodes (which hold the user-defined value) and the non-terminal nodes (which contain the language-defined types). For our representation of the abstract syntax tree, we concatenate the Prüfer sequence of the AST and list of terminal nodes. For the particular case of two-node AST, the Prüfer-sequence part of our representation is an empty list. Fig. 3.7 shows the Prüfer sequence of the AST. The Blue part of the sequence represents the internal nodes of the AST, while the green part of the sequence represents the terminal of the AST.

4. To Define a Context of the Code

To train a sequence-to-sequence model for tasks such as code summarization, a training example consists of a sequence of tokens of the code and the developer’s comment on the code. Since code comments are usually extracted (by human developers) from the lexical information of the source code, such as the method or variable names, it is important for a learning model to take advantage of such information. For example, Hu [HLX+20] in the Hybrid-DeepCom model proposed an attention-based dual encoder (Code Token encoder and SBT encoder) and a decoder model to generate comments of the code. Here the SBT will capture the syntactical information of the code while the code token will help capture the lexical information of the code. An AST (Fig. 3.2) representation in the Hybrid-DeepCom model is

\[
< \text{public int run Command ( boolean merge Error Into Output , String commands ) throws IOException , Interrupted Exception return runCommand ( merge Error Into Output , new ArrayList < String > ( Arrays . asList ( commands ) ) ) ; } >
\]
Chapter 3. Prüfer Sequence Generation

Figure 3.6: Prüfer sequence of the AST (Non-Terminal nodes represented by Blue nodes and the Terminal nodes are represented by Green Nodes)

To encode the entire code to learn the lexical information of the code is not efficient as many of the tokens in the code does not contribute to the lexical information of the code.

The prüfer sequence of an AST provides a natural way for us to model the context of a code. We define the context of a node in an AST to be the subset of its children that are terminal nodes, and we generate the set of the children using the Prüfer sequence. The context generated through the Prüfer sequence focuses on those tokens that are more significant in defining the context of the code. For example, the context of the tree (Fig 3.7) based on the Prüfer sequence (0 0 3 0 5 0 7 8 7 10 12 11 10 14) is {1 2 1 2 4 1 2 6 1 2 9 13 15}. We mapped this numerical sequence of context to the AST labeled sequence using the information stored at AST-Table. Context of the AST (Fig 3.2) after mapping is

```java
{ ( MethodDeclaration ( Annotation ) Annotation ( BasicType ) BasicType ( FormalParameter ( BasicType ) BasicType ) FormalParameter ( FormalParameter ( ReferenceType ) ReferenceType ) FormalParameter ( ReturnStatement ( MethodInvocation ( MemberReference ) MemberReference ( ClassCreator ( ReferenceType ( TypeArgument ( ReferenceType ) ReferenceType ) TypeArgument ) ReferenceType ( MethodInvocation ( MemberReference ) MemberReference ( MemberReference ) ) ) ) ) )

To encode the entire code to learn the lexical information of the code is not efficient as many of the tokens in the code does not contribute to the lexical information of the code.
Following are the examples of context generated for java codes using Prüfer sequence.

```java
private Socket getSocket(WebSocket conn)
{
  WebSocketImpl impl = (WebSocketImpl) conn;
  return ((SocketChannel)impl.getSelectionKey().channel()).socket();
}
```

(a) **Comment** Getter to return the socket used by this specific connection

(b) **Context Generated** get Socket Web Socket get Socket Web Socket Impl conn Web Socket Impl conn Web Socket Impl Web Socket Impl get Socket
protected CompletionStage <Result> onForbidden (RequestHeader request,String message)
{
    return CompletableFuture.completedFuture(Results.forbidden(views.html.defaultpages.unauthorized.render(request.asScala( ) ) ) ) ;
}

(a) **Comment** Invoked when a client makes a request that was forbidden
(b) **Context Generated** Result on Forbidden Request Header on Forbidden String on Forbidden request as Scala

public void broadcast ( String text,Collection <WebSocket> clients)
{
    if ( text == null || clients == null )
    {
        throw new IllegalArgumentException( );
    }
    doBroadcast(text,clients); }

(a) **Comment** Send a text to a specific collection of websocket connections
(param text the text to send to the endpoints)
(param clients a collection of endpoints to whom the text has to be send)
(b) **Context Generated** String broadcast Web Socket broadcast text text clients
    clients Illegal Argument Exception broadcast do Broadcast text clients text clients

public void withTransaction( Consumer<EntityManager> block )
{
    withTransaction( em ->{ block.accept(em); return null;});
}

(a) **Comment** Run a block of code with a newly created Entity Manager for the default Persistence Unit
(b) **Context Generated** Entity Manager with Transaction block accept em em block accept em
public final Self overrides(GuiceableModule ... modules )
{
  return newBuilder(delegate.overrides(Scala.varargs(modules)));
}

(a) **Comment** Override bindings using guiceable modules
(b) **Context Generated** overrides Guiceable Module overrides modules

public static Result status(int status, byte [ ]content)
{
  if(content==null)
  { throw new NullPointerException(STR_); }
  return new Result(status,new HttpEntity.Strict( ByteString. fromArray(content),
Optional.empty( ))); }

(a) **Comment** Generates a simple result with byte-array content
(b) **Context Generated** status int status byte status content content Null
    Pointer Exception Null Pointer Exception status Result status Result status
    Result status Optional empty Strict Optional empty content Optional empty

protected String removeLastOrderBy ( String sql )
{
  int ndx = StringUtil.lastIndexOfIgnoreCase(sql,STR_); 
  if(ndx != - NUM_)
  {
    int ndx2 = sql.lastIndexOf(sql,STR_); 
    if(ndx > ndx2)
    {
      sql = sql.substring( NUM_,ndx);
    }
  }
  return sql; }

(a) **Comment** Removes everything from last “order by”
(b) **Context Generated** remove Last Order By String remove Last Order By
    int sql sql int remove Last Order By ndx ndx int sql sql int sql ndx ndx sql
    sql remove Last Order By sql
Chapter 3. Prüfer Sequence Generation

```java
protected String removeLastOrderBy(String sql) {
    int ndx = StringUtil.lastIndexOfIgnoreCase(sql, STR_);
    if(ndx != - NUM_)
    {
        int ndx2 = sql.lastIndexOf(sql, STR_);
        if(ndx > ndx2)
        {
            sql = sql.substring(NUM_, ndx);
        }
    }
    return sql;
}
```

(a) **Comment** Removes everything from last “order by”

(b) **Context Generated** remove Last Order By String remove Last Order By int sql sql int remove Last Order By ndx ndx int sql sql int sql ndx ndx sql sql remove Last Order By sql

```java
protected String removeSelect(String sql) {
    int ndx = StringUtil.indexOfIgnoreCase(sql, STR_);
    if(ndx != - NUM_)
    {
        sql=sql.substring(ndx+NUM_);
    }
    return sql;
}
```

(a) **Comment** Removes the first ’select’ from the sql query

(b) **Context Generated** remove Select String remove Select int sql sql int remove Select ndx ndx sql ndx sql sql remove Select sql

We also point out that no special assumptions were made regarding AST or programming language, ensuring the same mechanism hold for other programming languages.
Chapter 4

Prüfer-Sequence-Based Learning Model For Code Summarization

To study the effectiveness of the Prüfer-sequence-based representation, we developed a deep-learning model for code summarization. The model maps a Java method to a summary of the method’s purpose in English. The training data are pairs of source code of the Java method and developers’ comments. The high-level structure of our model is depicted in Fig.4.1. It is a sequence-to-sequence (seq2seq) model [SVL14] in the encoder-decoder paradigm, where two separate encoders are used to learn from the structural information of an AST and from a structure-aware representation of the lexical tokens from the source code. An attention module, similar to the one used by [HLX+20], is used to combine the output of the two encoders into a context vector which is then used as the input to a standard decoder described in [SVL14] to output a code summary/comment in English.

Figure 4.1: Prüfer Based Learning Model for Code Summarization Task
4.0.1 Prüfer Sequence Encoder

The Prüfer Sequence Encoder is designed to learn from the structural information of the ASTs that are losslessly encoded in their syntactic Prüfer sequences. Gated Recurrent Units (GRUs), as discussed by [CvM14], are used to map the syntactic Prüfer sequence \((X = x_1, \ldots, x_n)\) of a computer program to a sequence of hidden states as follows:

\[
s_t = GRU(x_t, s_{t-1})
\]

In our implementation, lexical tokens labeling the terminal nodes are appended to the syntactic Prüfer sequence as part of the input to the encoder, resulting in an input length of at most \(2n - 3\). We observed in our initial experiments that the model Hybrid-DeepCom [HLX+20] with its SBT-based encoder replaced by our Prüfer Sequence Encoder already outperformed notably the original Hybrid-DeepCom Model and other baseline models. It turned out that the bottleneck to further improvement of such models is the design of the second encoder, the Source Encoder [HLX+20], that learns from source code tokens directly. This observation in our early investigation motivated us to design our own second encoder, which we call the Context Encoder, that exploits lexical information in a structure-aware way.

4.0.2 Context Encoder

The context encoder, also consisting of GRUs, is designed to learn from the collection of lexical tokens (i.e., user-defined and program-specific values in the source code) organized in a way that reflects the structural information of the AST.

For each node in an AST, we define its context to be the set of lexical tokens that label the node’s leaf child/children. A node with no leaf child has an empty context. The context of an AST is defined to be the union of the contexts of the AST nodes ordered in the same order as they appear in the syntactic Prüfer sequence. The context of an AST can be calculated from the AST’s Prüfer sequence (See Fig. 3.1). The context encoder maps the context \((X' = x'_1, \ldots, x'_n)\) defined in the above to a sequence of hidden states.

\[
s'_t = GRU(x'_t, s'_{t-1})
\]

The context of an AST defined in this way is a structure-aware sequence of lexical tokens because (1) the frequency of a lexical token is decided by the degree of the parent of the node the token labels; and (2) the order in which these tokens appear in the context is the same as the order of the parent nodes in the Prüfer sequence. As observed in our experiments, the use of the context encoder helps boost the performance of our learning model significantly. This is because, we believe, that the context we have designed helps amplify the learning-relevant lexical signals in the source code in a way that other models (such as those in [HLX+20]) cannot detect that by simply using the collection of entire tokens as they appear in the source code.
4.0.3 Attention

The attention model incorporated in an encoder decoder model will assist in focusing those input words that are more relevant at time step \( t \). The generation of each word is based on the classical attention method proposed by Bahdanau [BCB15]. In our model, the attention mechanism is designed to focus both the encoder, i.e., the Context Encoder, and the Prüfer Sequence Encoder, which is similar to the attention mechanism used by Hu [HLX+20]. For each target word \( y_i \) we define an individual \( c_i \) as the weighted sum of all the hidden states of both the encoders

\[
c_i = \sum_{j=1}^{m} \alpha_{ij} s_j + \sum_{j=1}^{m} \alpha'_{ij} s'_j
\]

where the \( \alpha \) and \( \alpha' \) are the attention distribution of Context encoder and Prüfer sequence encoder respectively. The \( \alpha_{ij} \) of the hidden state \( s_j \) is computed as

\[
\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{m} \exp(e_{ik})}
\]

and

\[
\alpha'_{ij} = \frac{\exp(e'_{ij})}{\sum_{k=1}^{m} \exp(e'_{ik})}
\]

where \( e_{ij} \) and \( e'_{ij} \) are computed as

\[
e_{ij} = a(h_{i-1}, s_j)
\]

and

\[
e'_{ij} = a(h_{i-1}, s'_j)
\]

where \( a \) is a alignment models [BCB15] which score how well the inputs around position \( j \) and the output at position \( i \) match.

4.0.4 Decoder

The decoder will generate the output \( y \) (comment of the code) by sequentially predicting the probability of a word \( y_i \) such that it is conditioned over the context vector \( c_i \) and its previous generated \( y_1, y_2, \ldots, y_{i-1} \)

\[
p(y_i|y_1, y_2, \ldots, y_{i-1}, x) = g(y_{i-1}, h_i, c_i)
\]

where \( g \) is used to estimate the probability of the word \( y_i \). The objective of the model is to reduce the cross-entropy [SVL14]

\[
h(y) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{n} \log p(y_j^i)
\]

where the \( N \) is the total number of the training instance and the \( n \) is the length of each target sequence and our model uses the optimization algorithm such as gradient descendental for optimizing the objective function.
4.1 Experiment Setup

We perform the code summarization task on two standard datasets a) the first dataset provided by the Hu [HLX+18b], we refer to this dataset as Dataset-1. Dataset-1 is collected from popular repositories in Github, is a parallel collection of Java code and comments. For dataset-1, we have a corpus of 68469 code and comment pairs for the code summarization task, and we split the dataset-1 into 8:1:1 for training, testing, and validation, respectively. The second dataset is the CodeXglue java dataset provided by the Microsoft [LGR+21]; we refer to this dataset as dataset-2. It is known for its high quality and complexity and is believed to be one of the most challenging datasets for deep learning approaches to program understanding and generation. For dataset-2, we have 163316 pairs of Java code and comments, and we split the dataset-2 into 8:1:1 for training, testing, and validation, respectively. And Fig 4.2 and 4.3 show the frequency distribution of Java code and comment length of the dataset-1 and Dataset-2 such that the x-axis shows the code and comment length while the y-axis shows the frequency of
the code and comment length. Tables 4.1 and 4.2 show the statistical summary of the Java code and comment for Dataset-1 and Dataset-2.

For building the vocabulary for the model, we follow the methods used by Hu [HLX+18a, SVL14]. For both dataset-1 and dataset-2, java code and comments are parsed into tokens using the Javalang and NLTK\(^2\) respectively. The numerical and string in the java code are replaced by the generic token \(<\text{NUM}\>\) and \(<\text{STR}\>\) respectively. Special tokens \(<\text{START}\>\) and \(<\text{EOS}\>\) are added to the decoder sequence during the training such that the \(<\text{START}\>\) represents the start of the sequence and the \(<\text{EOS}\>\) represents the end of the sequence. The out-of-vocabulary is represented by especial token \(<\text{UNK}\>\). The vocabulary size is set to 30000 tokens for code, comment, and code’s contex [HLX+18a, SVL14]. From Table 4.1 and 4.2, we observed that 88.42 % of the comments have less than 30 tokens and 89.48 % of the code has less than 200 tokens, so we set the maximum length of the code and comment

\(^2\text{https://www.nltk.org/}\)
for our experiment be 30 and 200 tokens respectively. For the AST sequence, we kept the maximum length of the sequence to be 500 tokens [HLX+18a, SVL14].

| Dataset Type | Mean | Mode | Median | SD | <200 Tokens |
|--------------|------|------|--------|----|-------------|
| Dataset-1    | 97.05| 16   | 64     | 119.77 | 89.48%      |
| Dataset-2    | 98.35| 42   | 69     | 83.64  | 89.29%      |

| Dataset Type | Mean | Mode | Median | SD | <30 Tokens |
|--------------|------|------|--------|----|------------|
| Dataset-1    | 16.1 | 7    | 11     | 16.91 | 88.42%     |
| Dataset-2    | 11.02| 7    | 9      | 7.34  | 97.33%     |

Our model uses one layered GRU with 256 dimensions of hidden state and 256-dimensional word embedding. The maximum iterations are 60 epochs and the beam width is set to 5. The learning rate is set to 0.5, and we clip the gradients norm by 5. The learning rate is decayed using the rate 0.99. The model uses the TensorFlow version 1.15, and we train our model on a single GPU of Tesla P100-PCIE-16GB with 25 GB RAM and 110GB disk.

4.1.1 Baseline

In our experiments, we compared the performance of our learning model with the following baseline models to empirically analyze the power, effectiveness, and efficiency of the proposed Prüfer-sequence-based representation.

1. **TL-CodeSum Model** [HLX+18b]. This is an NMT based code summarization method that uses API knowledge and source code tokens as the input in a sequence-to-sequence model.

2. **Hybrid-DeepCom Model** [HLX+20]. This sequence-to-sequence model for code summarization is one of the recent models that is designed to exploit structural information in the AST of a computer program. It uses two encoders: a source-token encoder and an SBT encoder. The SBT encoder uses a depth-first-traversal sequence representation of an AST as its input, and the code-token encoder is used to learn from the lexical information of the source code.

3. **Code2Seq Model** [ABLY19]. This is a deep-learning model for general code representation learning. Code2Seq uses the concatenation of the token sequences
4.1. Experiment Setup

along the paths between pairs of terminal nodes in an AST as its input representation to generate comments.

4. BFS-Hybrid-DeepCom. This model is based on the Hybrid-DeepCom Model [HLX+20] with the SBT representation of an AST replaced by a sequence representation constructed from a breadth-first-search (BFS) traversal of the AST. We included this model to help verify our claim that all the recently-proposed sequence representations are more or less arbitrary. As observed from our experiments (see next Section), this BFS-based model performs equally well (or even slightly better) than those recently proposed models.

5. Lexical-Token-Only Model. This basic attention-based seq2seq model has only one encoder that learns from the lexical information in the source code. We used this model to understand the importance of incorporating syntactic information (in the AST) in deep-learning approaches for code summarization. The parameter setting of the model is similar to that of the Hybrid-DeepCom model.

4.1.2 Metrics

To evaluate the effectiveness of different approaches, we use four widely-used machine translation metrics: two BLEU scores, the METEOR score, and ROUGE-L.

1. BLEU Score

The Bilingual Evaluation Understudy (BLEU) [PRWZ02] is a metric to check the quality of the machine-translated (candidate) text against that of the human-written (reference). The BLEU score is in the range of 0 to 100 such that the higher the value closer is the machine-generated text to that of human-written text. The primary task of a BLEU implementor is to compare n-grams of the candidate with n-grams of the reference translation and count the number of the matches. Here n-gram refers to the number of the words from the sequence matching the candidate and reference. For instance, a uni-gram will have one word from a sequence, and a bi-gram will have two words while matching between sentences. The BLEU score is computed as:

$$BLEU = BP \cdot \exp \left( \sum_{n=1}^{N} w_n \log p_n \right)$$

where $p_n$ is the ratio of length n subsequences in the candidate that is also in the reference, and we set N to 4 (which is the maximum number of a gram). $w_n$ is a positive weight such that $w_n = 1 / N$. BP is the brevity penalty, ensuring that a high-scoring candidate translation must match the reference translations in length, word choice, and word order. We compute the BP as:
4.1. Experiment Setup

\[ BP = \begin{cases} 
1 & \text{if } c > r \\
\frac{1}{e(1 - r/c)} & \text{if } c \leq r
\end{cases} \]

where \( c \) is the length of the candidate translation and \( r \) is the effective reference sequence length. In this study, we evaluate the approaches by Sentence-Level BLEU (S-BLEU) score computed by the NLTK\(^3\) (evaluating a candidate sentence against reference sentences) with a smoothing-4 method and Corpus-level BLEU (C-BLEU) calculated by “multi-bleu.perl”\(^4\) (Calculate a single corpus-level BLEU score for all the hypotheses and their respective references).

2. **ROUGE-L**

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a Machine Translation metric proposed by Chin [Lin04]. It measures the quality of a summary generated by the machine by comparing the number of overlapping units such as n-gram, word sequences, and word pairs between the computer-generated summary and the ideals summary created by humans. ROUGE-L is one of the four measures of the ROUGE where \( L \) stands for Longest Common Subsequence, and it computes the F-score, where F-score is calculated as the harmonic mean of the recall and precision value obtain from finding the longest common sequence of the texts.

Let us consider two summary sentences of \( X \) with \( m \) length and \( Y \) with \( n \) length such that \( X \) is a reference summary and \( Y \) is a candidate/Predicted summary sentences

\[ R_{LCS} = \frac{\text{LCS}(X,Y)}{m} \]
\[ P_{LCS} = \frac{\text{LCS}(X,Y)}{n} \]
\[ F_{LCS} = \frac{(1+\beta)^2 \cdot R_{LCS} \cdot P_{LCS}}{\beta \cdot R_{LCS} + P_{LCS}} \]

where \( \text{LCS}(X,Y) \) is the length of the longest common subsequence of \( X \) and \( Y \) and

\[ \beta = \frac{P_{LCS}}{R_{LCS}} \]

and the LCS-based F-measure is called the ROUGEL-L.

3. **METEOR**

Metric for Evaluation of Translation with Explicit ORdering (METEOR) [DL14] is a recall-oriented evaluation method. It evaluates translation hypotheses by aligning them to reference translations and calculating sentence-level similarity scores

\(^3\)https://www.nltk.org

\(^4\)http://www.statmt.org/moses/?n=Moses.SupportTools
4.2. Results

\[ \text{Score} = (1 - \text{Pen}) F_{\text{mean}} \]

where penalty (Pen) is the ratio number of chunks (ch), where a chunk is defined as a series of matches that is contiguous and identically ordered in both sentences and number of matches (m, averaged over hypothesis and reference) and calculated as

\[ \text{Pen} = \gamma \cdot \left( \frac{ch}{m} \right)^{\beta} \]

\[ F_{\text{mean}} = \frac{P \cdot R}{\alpha \cdot P + (1 - \alpha) \cdot R} \]

where F-mean measures the parameterized harmonic mean of Precision and Recall. The parameters \( \alpha, \beta \) and \( \gamma \) are tuned to maximize correlation with human judgments. For English Denkowski [DL14] set the value of \( \alpha \) to 0.85, \( \beta \) to 0.20 and \( \gamma \) to 0.60.

**Precision** The classical definition of precision is that it is the ratio of correctly predicted positive observation to the total predicted positive observation. We calculate the precision using the method proposed by Denkowski [DL14]. This method gives different weights for content and function words. Context words provide the most important words in the sentences, while the function words are the grammar that stitches the context words. The content and function words in the generated comment and reference are \((h_c, h_f)\) and \((r_c, r_f)\) respectively. For each of the matches \(m_i\), count the number of content and function words covered by matches of this type in the hypothesis \((m_i(h_c), m_i(h_f))\) and reference \((m_i(r_c), m_i(r_f))\)

\[ P = \sum_i w_i \frac{\delta \cdot m_i(h_c) + (1 - \delta) \cdot m_i(h_f)}{\delta |h_c| + (1 - \delta) |h_f|} \]

where \(\delta\) is word weight of the content-function and is set to 0.75 according to Denkowski [DL14].

**Recall** Following the precision, we choose a weighted Recall to measure models

\[ R = \sum_i w_i \frac{\delta \cdot m_i(r_c) + (1 - \delta) \cdot m_i(r_f)}{\delta |r_c| + (1 - \delta) |r_f|} \]

4.2 Results

In Tables 4.3 and 4.4, we summarize the experiment results on the effectiveness of our Prüfer-based learning model and the baseline models for code summarization task.
on two public Java datasets: a) Dataset-1 [HLX$^+18b$] and b) Dataset-2 [LGR$^+21$]. We note that the performance scores for Dataset-2 is much lower, but this is not a surprise—as we mentioned in the previous section, this dataset is believed to be one of the most challenging dataset for deep learning approaches to program understanding and generation. A detailed discussion can be found on CodeXGLue’s project webpage (https://microsoft.github.io/CodeXGLUE/). Also, note that in Table 4.4 (for Dataset-2), we do not have a row for the model TL-CodeSum because Dataset-2 does not provide the API knowledge required to train the model. As we can see, the model that uses our Prüfer Sequence Encoder and Hu’s Source Encoder (second last rows in both Tables 4.3 and 4.4) has already had notable improvement over the baseline models. The improvement by our complete Prüfer-sequence-based model (using our Prüfer Sequence Encoder and our Context Encoder) is even more significant, especially on Dataset-2 (last rows in both Tables 4.3 and 4.4.)
### 4.2. Results

Table 4.3: Effectiveness of Models based on Machine Translation metrics for Dataset-1 and value inside the bracket shows the performance improvements in percentage compare to the baseline models

| Model                                      | S-BLEU | C-BLEU | METEOR | ROUGE-L |
|--------------------------------------------|--------|--------|--------|---------|
| Lexical-Token-Only Model                   | 36.21  | 27.30  | 19.01  | 40.78   |
| Code2Seq                                   | 20.72  | 4.56   | 10.21  | 20.63   |
| TL-CodeSum                                 | 37.20  | 28.43  | 19.64  | 41.29   |
| BFS-Hybrid-DeepCom                         | 37.98  | 29.08  | 19.72  | 41.03   |
| Hybrid-DeepCom                             | 38.19  | 29.28  | 19.87  | 41.15   |
| **Our Model (Prüfer Encoder + Hu’s Source Encoder)** | **38.38 (0.5%)** | **29.43 (0.5%)** | **20.13 (1.3%)** | **41.82 (1.3%)** |
| **Our Model (Prüfer Encoder + Context Encoder)** | **39.67 (3.3%)** | **31.01 (5.7%)** | **21.01 (5.6%)** | **43.45 (5.1%)** |

Table 4.4: Effectiveness of Models based on Machine Translation metrics for Dataset-2 and value inside the bracket shows the performance improvements in percentage compare to the baseline models

| Model                                      | S-BLEU | C-BLEU | METEOR     | ROUGE-L   |
|--------------------------------------------|--------|--------|------------|-----------|
| Lexical-Token-Only Model                   | 9.21   | 3.07   | 7.96       | 19.84     |
| Code2Seq                                   | 2.27   | 0.30   | 3.5        | 12.23     |
| BFS-Hybrid-DeepCom                         | 13.41  | 3.47   | 7.29       | 20.42     |
| Hybrid-DeepCom                             | 15.02  | 3.7    | 8.27       | 18.01     |
| **Our Model (Prüfer Encoder + Hu’s Source Encoder)** | **15.50 (3.15%)** | **3.85 (3.97%)** | **8.9 (6.925%)** | **20.79 (1.8%)** |
| **Our Model (Prüfer Encoder + Context Encoder)** | **16.15 (7.02%)** | **4.49 (19.29%)** | **9.72 (15.05%)** | **24.73 (19.09%)** |
In the rest of this section, we discuss our observations from the experiments and analyze the results to understand the power, effectiveness, efficiency, and the robustness (against the code length) of our Prüfer-sequence-based representation. These observations and their analyses are solid evidence, supporting our belief that our Prüfer-sequence-based representation makes it possible (or much easier) for learning models to exploit information in the training examples that are hard, if not impossible, for other recently-proposed learning models to detect.

**Power and Effectiveness of Prüfer-Sequence Representation of AST.**

As shown in the second last rows in Tables 4.3 and 4.4, our Prüfer Sequence Encoder and Hu’s Source Encoder [HLX+20], used to learn the lexical information of code by taking the entire code as input to the Source Encoder (second last rows in both Tables 4.3 and 4.4) has already had notable improvement over the baseline models, with the average performance improvement being 0.9% for Dataset-1 and 3.96% for Dataset-2. We attribute the performance improvement to the properties of the Prüfer sequence representation discussed in the previous Section (Abstract Syntax Trees and Prüfer Sequences): a concise and lossless encoding that quantifies the “importance” of syntactic tokens and preserves their structural roles in describing the source code. This is further supported by the relatively poor performance of the recently-proposed general model, Code2Seq, that uses a lossy encoding with the input length cubic to the size of the AST in the worst case. Note that the performance of the Code2Seq Model is worse than the model that does not make use of any structural information of the AST (first rows in both tables).

We can also see that the performance of the two versions of the Hybrid-DeepCom Model based on respectively the BFS sequence and the SBT sequence are comparable on both datasets, justifying our claim in the introduction that in such traversal-based sequence representation recently proposed, the order of the tokens in a sequence is largely arbitrary and only partially captures the structure of an AST.

**Importance of Structure-Aware Context Sequence of Lexical Tokens.**

As shown in the last row in both Tables 4.3 and 4.4, the use of the Context Encoder boosted the performance of our model. The average performance improvement over baseline models is increased from 0.9% to 5% for Dataset-1 and 3.96% to 15.11% for Dataset-2.

Considering that both of the two recently-proposed deep-learning models (Code2Seq and Hybrid-DeepCom, the second rows and the third last rows in both tables) are also designed to make use of structural information of an AST as well as lexical tokens from the source code, the significant performance gain of our model is best interpreted by the fact that the “context” sequence defined in our model as the input to the Context Encoder is structure-aware. The difference between our context sequence and the those used in the Hybrid-DeepCom model and the Code2Seq model is that in our context sequence, the frequency of a lexical token from the source code is decided by the degree
4.2. Results

of its parent node in the AST, whereas Hybrid-DeepCom and Code2Seq treat tokens from the source code equally regardless of their role and significance in the program.

4.2.1 Performance over Source Code of Different Lengths

The histogram in Fig. 4.4 shows the frequency distribution of the code length (token count). From Fig. 4.4, we can say that more than 90% of the java method has a length of 25 to 250 tokens. Therefore we will consider the code length between 25 to 250 tokens while measuring the performance of three learning models: Our Model, Hybrid-DeepCom, BFS-Hybrid-DeepCom.

![Frequency Distribution of Code Length](image)

Figure 4.4: Frequency Distribution of Code Length

We observe (Fig.4.5) that for all the three models, the performance decreases as the code length increases. When the code length is between 25 to 50 tokens, the model achieves optimal performance. We note that our Prüfer-sequence-based model performs better than the other two baseline models regardless of the code lengths. For the Java method with 150 or more tokens, the Prüfer sequence-based model had a clear edge over the other two methods, suggesting that it is more robust against the increase of code length. To test the statistical significance of the result, we conduct a t-test (Independent
4.2. Results

A t-test) between the Prüfer-sequence-based model and SBT model, and we get a p-value of 0.0004, showing results are significant. Also for Prüfer-sequence-based model and BFS model we get a p-value of 0 (as value is very low) showing a significant result.

The correlation coefficient between the BLEU score and code length is the smallest for the Prüfer-sequence-based model (-0.037), while for BFS and SBT, it is -0.16 and -0.19, respectively. This indicates that the (negative) correlation between the performance of models and the code length is much weaker for our model than the other two models.

![Figure 4.5: BLEU score for different method lengths](image)

4.2.2 Flexibility of Prüfer Sequence

We incorporate the Prüfer sequence into two existing models a) Hybrid-DeepCom Model by Hu [HLX+20] where we replace the SBT with Prüfer sequence and b) TL-CodeSum [HLX+18b] where we replace the code itself with that of the Prüfer sequence.

1. In the Hybrid DeepCom model, the empirical result from Table 4.5 shows that the performance of the Hybrid DeepCom model with the Prüfer sequence is similar to that of the Hybrid DeepCom model with SBT (slight improvement across all metrics). This result can be attributed to the fact that both SBT and the Prüfer sequence use the syntactic type of tokens to represent the sequence.

2. In the TL-CodeSum model, the empirical result from Table 4.5 shows that the performance of the Hybrid DeepCom model decreases as we replace the code
with the Prüfer sequence. Since the TL-CodeSum model [HLX+18b] relied on the semantic value (actual code as input to model) for the comment generation, thus replacing the semantic value with the syntactic tokens of the Prüfer sequence result in the drop in the performance.

Table 4.5: Flexibility of Prüfer sequence for Dataset-1

| Model                   | S-BLEU | C-BLEU | METEOR |
|-------------------------|--------|--------|--------|
| TL-CodeSum              | 37.20  | 28.43  | 19.64  |
| Hybrid-DeepCom          | 38.19  | 29.28  | 19.87  |
| TL-CodeSum(Prufer)      | 29.58  | 20.78  | 15.54  |
| Hybrid-DeepCom (Prufer) | 38.38  | 29.43  | 20.13  |

From these two observations, we can see that the Prüfer sequence is flexible and can be used with those existing models where the input to the model is the syntactic type of tokens.

4.2.3 Efficiency of the Prüfer Sequence in encoding the AST of the code

The input dimension to the seq-to-seq deep learning model depends on the encoding scheme of the code. The lower the dimension is, the faster it is to complete one epoch of training. There is, of course, a tradeoff among the effectiveness/ability of a model, its input dimension, and the training time. An ideal encoding is the one that preserves as much as possible the structural information and has a short length.

Table 4.6: Statistics of the AST Representation Methods for Dataset-1

| Method       | Mean | Mode | Median | <100 Tokens | <150 Tokens | <200 Tokens |
|--------------|------|------|--------|-------------|-------------|-------------|
| Prufer Sequence | 100.81 | 8    | 82     | 67.74%      | 81.11%      | 88.06%      |
| SBT          | 193.71 | 24   | 124    | 41.42%      | 58.01%      | 68.77%      |
| BFS          | 70.21  | 8    | 44     | 80.02%      | 89.6%       | 94.16%      |

Our experiments confirm that our Prüfer sequence representation encodes the structure of an AST losslessly and is more concise than other representations indeed requires less time to train. Fig. 4.6 summarizes our observation on the time required to complete different training epochs for three learning models: Our Model,
Hybrid-DeepCom, BFS-Hybrid-DeepCom. Among the models, BFS-Hybrid-DeepCom uses the shortest sequence representation (70.21 on average over the training data), and Hybrid-DeepCom has the longest representation (193.91 on average). The average length of our Prüfer sequence representation is 100.81. It is a surprise to observe that BFS-Hybrid-DeepCom (a model we customized from Hybrid-DeepCom using a straightforward and much shorter breadth-first-search-based representation) requires less training time but has a comparable performance with Hybrid-DeepCom that is based on a carefully designed and more sophisticated representation. While our model requires more time to train than BFS-Hybrid-DeepCom (as expected but not by much), the performance gain of our model is significant.

![Figure 4.6: Time required to train Prüfer Based Learning Model for 60 epoch using different AST representation.](image)

4.2.4 Comment Generation Using Different AST Representation Methods

One advantage of the Prüfer based code representation learning model is that it generates the comment based on the context of the code. The code’s context ensures that tokens relevant to the comments are sent as input to the model and generate quality comments.

The general approach of the comments relied either on the manually crafted templates (which is time-consuming, and the quality of keywords depends on the quality of a given Java method) or the IR-based approach (relying on finding similar code snippets). Hu
4.2. Results

[HLX+20] uses the flat sequence of tokens to learn the lexical information to generate the comments, which is inefficient as most of the tokens in the code are not part of the comment generation.

In our approach, the Prüfer sequence ensures that the context of the code focuses on those terminal tokens relevant in generating the comments. The model learns the lexical information of the code through the context of the code, and the attention mechanism helps to align the context of the code and natural language words. Following are the examples of the quality of comments generated by the Prüfer sequence-based learning model.

```java
public void print ( final int width , @NonNull final PrintStream aPS ) {  
    final SystemOutAlignRight ar = new SystemOutAlignRight ( aPS ) ;
    for ( int r = NUM_ ; r < m_nRows ; ++ r )
        { ar.print( STR_, NUM_ );
          ar.print(r+NUM_, NUM_);
          ar.print(STR_,NUM_);
          for(int c=NUM_;c<m_nCols;++c) {
              ar.print(m_aValues[r][c], width );
          }
          ar.println ( ) ; }
```

1. **Reference** Print the matrix values

2. **Prüfer Sequences** Print the contents of the given 2D column to the given PrintStream

3. **SBT** Print the contents of the given at the beginning at the end

4. **BFS** Prints the size of the space

5. **Code2Seq** a contact from set to default to as from

```java
final void signalWork ( WorkQueue [] ws , WorkQueue q ) {  
    long c ;
    int sp , i ;
    WorkQueue v ;
    Thread p ;
    while ( ( ( c = ctl ) < NUM_ ) ){
        if ( ( sp = ( int ) c ) == NUM_ )
            { 
                if ( ( c & ADD_WORKER ) != NUM_ ) tryAddWorker ( c ) ;
                break ;
            } if ( ws == null )
        break ;
    }
```

58
4.2. Results

```java
if ( ws.length <= ( i = sp & SMASK ) )
break;
if ( ( v = ws[ i ] ) == null )
break;
int vs = ( sp + SS_SEQ ) & ~ INACTIVE ;
int d = sp - v.scanState;
long nc = ( UC_MASK & ( c + AC_UNIT ) ) | ( SP_MASK & v.stackPred ) ;
if ( d == NUM_ && U.compareAndSwapLong( this , CTL , c , nc ) ) { v.scanState = vs ;
if ( ( p = v.parker ) != null ) U.unpark( p ) ; break ;
}
if ( q != null && q.base == q.top ) break ;
}
```

1. **Reference** Tries to create or activate a worker if too few are active

2. **Prüfer Sequences** Tries to try and enter a worker if it is available and if so

3. **SBT** Final external auth in avoid cleaning up in queue

4. **BFS** Uses to create worker heartbeat a worker and wait for too much

5. **Code2Seq** to a has and a task of to

```java
public void registerDefaultProcessor ( RemotingProcessor<? > processor ) {
if ( this.defaultProcessor == null )
{
this.defaultProcessor = processor;
} else {
throw new IllegalStateException( STR_ + this.defaultProcessor.getClass() );
}
}
```

1. **Reference** Register the default processor to process command with no specific processor registered

2. **Prüfer Sequences** Sets the default constructor to be used when creating a new processor

3. **SBT** Sets the thread pool used by this parser

4. **BFS** Sets the processor processor the processor processor fallback processor
5. **Code2Seq** is a tool for...

```java
public < T > List < T > getList ( String path , Class < T > genericType ) {
    if ( genericType == null ) {
        throw new IllegalArgumentException ( STR_ ) ;
    }
    final List < T > original = get ( path );
    final List < T > newList = new LinkedList < T > ( );
    if ( original != null ) {
        for ( T t : original ) {
            T e ;
            if ( t instanceof Map &&
                 !genericType.isAssignableFrom ( Map . class ) ) {
                String str = objectToString ( t );
                e = ( T ) jsonStringToObject ( str , genericType );
            } else {
                e = ObjectConverter.convertObjectTo ( t , genericType ) ;
            }
            newList.add ( e );
        }
    }
    return Collections.unmodifiableList( newList );
}
```

1. **Reference** Get the result of an Object path expression as a list
2. **Prüfer Sequences** Returns a list of objects from the given path
3. **SBT** Returns a list of values from the generic class with the given generic class
4. **BFS** Get a list of values from the given generic path
5. **Code2Seq** list and object object value

```java
public static ServerWebExchangeMatcher pathMatchers ( HttpMethod method , String ... patterns ) {
    List < ServerWebExchangeMatcher > matchers = new ArrayList < > ( patterns.length );
    for ( String pattern : patterns ) {
        matchers.add(
            new PathPatternParserServerWebExchangeMatcher ( pattern , method ) ) ;
    }
    return new OrServerWebExchangeMatcher ( matchers );
}
```
4.2. Results

1. **Reference** Creates a matcher that matches on the specific method and any of the provided patterns

2. **Prüfer Sequences** Creates a set of matcher classes from the given methods

3. **SBT** Create a set of predicates from the given method

4. **BFS** Creates a filter from the given list of parameter types

5. **Code2Seq** creates a new instance of method to for in and and and and

```java
public Buffer append ( final Buffer buffer )
{
    if ( buffer.list.isEmpty ( ) )
    {
        return buffer;
    }
    list.addAll( buffer.list);
    last = buffer.last;
    size += buffer.size;
    return this;
}
```

1. **Reference** Appends other buffer to this one

2. **Prüfer Sequences** Appends the record to the buffer

3. **SBT** Appends the buffer to the buffer

4. **BFS** Appends the contents of the buffer to the given buffer

5. **Code2Seq** buffer value the

```java
public void setGastgewTyp ( Gastgewerbe . GastgewTyp value )
{
    this . gastgewTyp = value ;
}
```
4.2. Results

1. **Reference** Sets the value of the gastgewTyp property

2. **Prüfer Sequences** Sets the value of the property

3. **SBT** Sets the value of the attribute

4. **BFS** Gets the value of the property

5. **Code2Seq** set for and value

```java
public void setGastgewTyp ( Gastgewerbe . GastgewTyp value )
{
    this . gastgewTyp = value ;
}
```

1. **Reference** Sets the value of the gastgewTyp property

2. **Prüfer Sequences** Sets the value of the property

3. **SBT** Sets the value of the attribute

4. **BFS** Gets the value of the property

5. **Code2Seq** set for and value

```java
public GeoPackageGeometryData getGeometry ( )
{
    GeoPackageGeometryData geometry = null ;
    int columnIndex = getTable ( ).getGeometryColumnIndex ( );
    int type = getType ( columnIndex );
    if( type != FIELD_TYPE_NULL )
    {
        byte [ ] geometryBytes = getBlob ( columnIndex );
        if ( geometryBytes != null )
        {
            geometry = new GeoPackageGeometryData ( geometryBytes );
        }
    }
    return geometry;
}
```

1. **Reference** Get the geometry@return geometry data
4.2. Results

2. **Prüfer Sequences** This method returns all of the items in the database that are read from the file.

3. **SBT** Gets the value of the property.

4. **BFS** Load the feature geometry.

5. **Code2Seq** name name and to to
Chapter 5

Conclusions

Automated code comprehensions increasingly rely on deep learning models such as neural machine translation (NMT) for tasks such as code summarization, program property predictions, and code classification. Earlier models treat the code as a flat sequence of tokens and pass it as input to the code. However, studies [HLX+18a, ABLY19] show that such straightforward representation fails to capture the syntactic information of the code, which plays a vital role in defining the functionality of the code. Several methods were proposed to capture the syntactic information of the code using the abstract syntax tree, for instance, classical traversal methods (Breadth-First Search and Depth First Search), Structure-Based Traversal (SBT) [HLX+18a], and paths between the terminals nodes of the AST [ABLY19]. These representations are lossy, and the choice of the tokens is arbitrary in some methods.

In this work, we proposed a concise and effective representation scheme that can be used in sequence-to-sequence models for code representation learning. By encoding structural information of abstract syntax trees of computer programs, our Prüfer-sequence-based representation makes it possible (or much easier) to develop sequence-to-sequence learning models to exploit automatically and selectively lexical and syntactic signals that are hard, if not impossible, for other recently-proposed sequence-to-sequence learning models to detect. The context of the code generated from the Prüfer-sequence has a natural advantage over the flat sequence of code as the frequency of the tokens in Prüfer-sequence is one less than the degree of the node in the tree. Thus a context based on the Prüfer-sequence ensures that the context sequence focuses on those tokens which are more important in understanding the lexical information of the code, which improved the performance of the model.

[ACRC20] investigated the possibility of incorporating AST information in their transformer-based model and concluded that AST information does not provide any help. Our studies in this paper suggest that it depends on how the (hierarchical) AST information is encoded and used in a learning model. It is a very interesting future work to study how our Prüfer-sequence-based encoding of ASTs can be used in a transformer-based model (such as the one in [ACRC20]) to decrease the model’s complexity and improve its effectiveness.

While the model we developed and the experiments conducted are on the task of code summarization for a particular program language, no assumptions were made about the programming language and the specification and the format of the abstract syntax trees, making our approach language independent and potentially applicable to other tasks in program comprehension and in any application domains of sequence-to-sequence learning.
Chapter 5. Conclusions

models where sequential and hierarchical signals both exist in the training data.

Thence, the Prüfer-sequence is a lossless, concise and effective encoding representation of the code. And from our empirical study, we provide a proof of concept that Prüfer-sequence can be used for the linear encoding of the AST of the code for learning the code representation.
Bibliography

[AA21] Norah Alsaaran and Maha Alrabiah. Classical arabic named entity recognition using variant deep neural network architectures and bert. IEEE Access, 9:91537–91547, 2021. → pages 2

[ABLY19] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In International Conference on Learning Representations, 2019. → pages 5, 10, 11, 25, 26, 35, 47, 64

[ACRC20] Wasi Uddin 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, pages 4998–5007. Association for Computational Linguistics, 2020. → pages 24, 25, 64

[And19] Rick Anderson. Rnn,talking about gated recurrent unit - dlbt: Deep learning benchmark tool. DLBT — Deep learning benchmark tool, Aug 2019. → pages 19

[ATGW15] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In Proceedings of the 32nd International Conference on Machine Learning, volume 37, pages 2123–2132. Proceedings of Machine Learning Research, 2015. → pages 24

[AZLY18] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, page 404–419. Association for Computing Machinery, 2018. → pages 2, 24, 35

[BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In 3rd International Conference on Learning Representations, 2015. → pages 22, 44

[BGB+17] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In Proceedings International Conference on Learning Representations 2017. OpenReviews.net, 2017. → pages 24
[BRV16] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code. In Proceedings of The 33rd International Conference on Machine Learning, volume 48, pages 2933–2942. Proceedings of Machine Learning Research, 2016. → pages 24

[CGCB14] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In NIPS 2014 Workshop on Deep Learning, December 2014, 2014. → pages 17, 19

[CvM14] Kyunghyun Cho and Bart van Merrienboer. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1724–1734, 2014. → pages 43

[DL14] Michael Denkowski and Alon Lavie. Meteor universal: Language specific translation evaluation for any target language. In Proceedings of the Ninth Workshop on Statistical Machine Translation, pages 376–380. Association for Computational Linguistics, 2014. → pages 49, 50

[HAM10] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, page 223–226. Association for Computing Machinery, 2010. → pages 24

[HIGH16] Tjalling Haije, Bachelor Opleiding Kunstmatige Intelligentie, E Gavves, and H Heuer. Automatic comment generation using a neural translation model. Inf. Softw. Technol, 55(3):258–268, 2016. → pages 2, 24

[HLWM20] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. Improved automatic summarization of subroutines via attention to file context. ACM, 2020. → pages 24

[HLX+18a] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension, ICPC ‘18, page 200–210. Association for Computing Machinery, 2018. → pages 5, 8, 24, 25, 26, 35, 36, 46, 47, 64

[HLX+18b] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. In Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI’18, page 2269–2275. AAAI Press, 2018. → pages 45, 47, 51, 55, 56

[HLX+20] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. Empirical Software Engineering, Springer, 25(3):2179–2217, 2020. → pages 3, 25, 36, 42, 43, 44, 47, 48, 53, 55, 58
[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. → pages 17

[IKCZ16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke 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. Association for Computational Linguistics, 2016. → pages 5, 7, 24, 35, 36

[KMCA06] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987, 2006. → pages 24

[Kos19] Simeon Kostadinov. Understanding encoder-decoder sequence to sequence model. *Towards Data Science Article*, Nov 2019. → pages 21

[LGR+21] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021. → pages 45, 51

[LHWM20] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 184–195, 2020. → pages 25

[Lin04] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, 2004. Association for Computational Linguistics. → pages 49

[LJM19] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE ’19, page 795–806, 2019. → pages 24

[LSLW16] Yang Liu, Chengjie Sun, Lei Lin, and Xiaolong Wang. Learning natural language inference using bidirectional lstm model and inner-attention. *arXiv preprint arXiv:1605.09090*, 2016. → pages 2

[LVD06] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006. → pages 2, 24

68
[MAS+13] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In 2013 21st International Conference on Program Comprehension (ICPC), pages 23–32, 2013. → pages 24

[Phi20] Michael Phi. Illustrated guide to lstm’s and gru’s: A step by step explanation. Medium, Jun 2020. → pages 18

[Prü18] Heinz Prüfer. Neuer beweis eines satzes über permutationen (a new prof. of a theorem on permutations). Archiv der mathematik und Physik, 3:27, 1918. → pages 12

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pages 311–318. Association for Computational Linguistics, 2002. → pages 48

[RBV16] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, page 731–747. Association for Computing Machinery, 2016. → pages 24

[RBVK16] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, page 761–774. Association for Computing Machinery, 2016. → pages 24

[SHM+10] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In Proceedings of the IEEE/ACM international conference on Automated software engineering, pages 43–52, 2010. → pages 2, 24

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS’14, page 3104–3112. MIT Press, 2014. → pages 20, 42, 44, 46, 47

[TSK+20] Amirsina Torfi, Rouzbeh A Shirvani, Yaser Keneshlou, Nader Tavaf, and Edward A Fox. Natural language processing advancements by deep learning: A survey. Turkish Journal of Computer and Mathematics Education, 2020. → pages 1

[TSM15] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In Proceedings of the 53rd Annual Meeting of the Association
Bibliography

for Computational Linguistics and the 7th International Joint Conference
on Natural Language Processing (Volume 1: Long Papers), pages 1556–1566.
Association for Computational Linguistics, 2015. → pages 25

[Tur50] A. M. Turing. Computing machinery and intelligence. Mind, 59(236):433–
460, 1950. → pages 1

[Wes00] Douglas B. West. Introduction to Graph Theory. Prentice Hall, 2000. →
pages 12

[WY18] Lili Wang and Hongwu Yang. Tibetan word segmentation method based
on bilstm_crf model. In 2018 International Conference on Asian Language
Processing (IALP), pages 297–302. IEEE, 2018. → pages 2

[WZY+18] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu,
and Philip S. Yu. Improving automatic source code summarization via deep
reinforcement learning. pages 397–407, 2018. → pages 24

[XBL+18] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan,
and Shanping Li. Measuring program comprehension: A large-scale field
study with professionals. In Proceedings of the 40th International Conference
on Software Engineering, page 584. Association for Computing Machinery,
2018. → pages 23, 24

[XMQ+19] Guixian Xu, Yueting Meng, Xiaoyu Qiu, Ziheng Yu, and Xu Wu. Sentiment
analysis of comment texts based on bilstm. IEEE Access, 7:51522–51532,
2019. → pages 2

[YJL+16] Jun Yin, Xin Jiang, Zhengdong Lu, Lifeng Shang, Hang Li, and Xiaoming
Li. Neural generative question answering. In Proceedings of the Workshop
on Human-Computer Question Answering, pages 36–42. Association for
Computational Linguistics, 2016. → pages 2

[ZWZ+19] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and
Xudong Liu. A novel neural source code representation based on abstract
syntax tree. In Proceedings of the 41st International Conference on Software
Engineering, page 783–794. IEEE Press, 2019. → pages 25

[ZWZ+20] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu.
Retrieval-based neural source code summarization. In Proceedings of the
ACM/IEEE 42nd International Conference on Software Engineering, page
1385–1397. Association for Computing Machinery, 2020. → pages 2, 24
import json
import csv

def Prufer_(cur_root_id, node_list):
    cur_root = node_list[cur_root_id]
    list1=[]
    list2=[]
    tmp_list=[]
    tmp_list.append('(')
    val=" 

    if 'children' in cur_root:
        str1 = cur_root['type']
    elif 'children' not in cur_root:
        str1 = cur_root['type']+' '+cur_root['value']

    tmp_list.append(str1)
    if 'children' in cur_root:
        if 'value' in cur_root:
            val=cur_root['value']
            #print(val)
        elif 'value' not in cur_root:
            val="XXX"
            #print(val)
    chs = cur_root['children']
    with open("train-prufer.csv", "a") as f:
        writer = csv.writer(f)
        for x in chs:
            writer.writerow((cur_root['id'],x,cur_root['type'],val))

    for ch in chs:
        tmp_list.extend(Prufer_(ch, node_list))
    elif 'children' not in cur_root:
        with open("train-prufer.csv", "a") as f1:
            writer1 = csv.writer(f1)
            writer1.writerow((cur_root['id'],cur_root['id'],cur_root['type']+' '+cur_root['value'],
                                cur_root['value']))
        tmp_list.append(')')
        tmp_list.append(str1)
def get_prufer_structure(ast_file, out_file):
    count = 1
    with open(ast_file, 'r') as ast_file:
        with open(out_file, 'w+') as out:
            asts = ast_file.readlines()
            for a in asts:
                a = json.loads(a)
                # print(set(range(0, len(a))))
                for i1 in a:
                    if 'children' in i1:
                        ko = i1['children']
                        for i in ko:
                            check2.add(i)
                    else:
                        check1.append(i1['children'])
                if set(check2) == set(range(0, len(a))):
                    count = count + 1
                    tmp_list = Prufer_(0, a)
                    out.write(' '.join(tmp_list) + '
')
                    print(count)
                else:
                    print("not equal")
                    val = input("value ")
                    get_prufer_structure("train_code", "train.token.ast2640.txt")

get_prufer_structure("train_code", "train.token.ast2640.txt")
import networkx as nx
import re

def prufercode(list1, list2, list3, count):
    try:
        if(len(list1)>0):
            list4=[]
            T = nx.Graph()
            T.add_nodes_from(list1)
            T.add_nodes_from(list2)
            edgeList = list(zip(list1,list2))
            edgeList1 = list(zip(list1,list2,list3))
            T.add_edges_from(edgeList,label = 1)
            T.remove_edges_from(nx.selfloop_edges(T))
            #degrees = [val for (node, val) in T.degree()]

            list5=[]
            list6=[]

            for (node, val) in T.degree():
                if val == 1:
                    list5.append(node)

            #print(T.number_of_nodes())

            if T.number_of_nodes() <= 2:
                #k=nx.bfs_tree(T, 0)# for bfs
                #print(k)
                k=[1,0,0,1] #for prufer code

            elif T.number_of_nodes() > 2:
                #k=nx.bfs_tree(T, 0)# for bfs
                k=nx.to_prufer_sequence(T) #for prufer code

            n=len(T)

            #print(count,set(T),set(range(n)))
fruit_dictionary = dict(zip(list1, list3))

# print(fruit_dictionary)

for j in list5:
    finalstring=[]
    j1=fruit_dictionary[j]
    temp_terminal=j1.split()

    if (len(temp_terminal)>1):
        temp_first=temp_terminal[0]
        finalstring.append(temp_first)

        temp_second=temp_terminal[1]
        if (temp_second != "STR_" and temp_second != "NUM_" and temp_second != "BOOL_"
            and temp_second != "XXX"):
            if (temp_second.isupper() != True):
                temp_second=' '.join(re.split(',|\.',temp_second))
                temp_second= re.split('_+', temp_second)
                for count26400 in range(len(temp_second)):
                    if (temp_second[count26400].isupper() != True):
                        finalstring.append(' '.join(re.sub( r"([A-Z])", r" \1", temp_second[count26400]).split()))

    # print(finalstring)
    list6.append(' '.join(finalstring))

    # list6.append(j1)

for i in k:
    z1=fruit_dictionary[i]
    list4.append(z1)

    # print(list4)
    list4.extend(list6)  # join the prufer code and terminal code
    # print(list4)
```python
# print(list4)
print(count)
with open("train-prufer-sequence-26NOV","a") as outfile:
    outfile.write(' '.join(map(str,list4))+'\n')
except Exception as e:
    n=len(T)
    print(count,set(T),set(range(n)))
    print(e)
val=input("enter the value ")
import pandas as pd
list1=[]
list2=[]
list3=[]
mid=[]
fd=pd.read_csv("train-prufer.csv",header=None)
pre=0
count=0
for i,j,k in zip(fd[0],fd[1],fd[2]):
    if(pre <= i):
        list1.append(i)
        list2.append(j)
        list3.append(k)
        pre=i
    elif(pre > i):
        prufercode(list1,list2,list3,count)
        count=count+1
        list1=[]
        list2=[]
        list3=[]
        list1.append(i)
        list2.append(j)
        list3.append(k)
        pre=0
prufercode(list1,list2,list3,count)
```
import networkx as nx
import re
from networkx.algorithms.traversal.depth_first_search import dfs_tree

def neighbour(t, item, st1, list1, list2):
    for n1 in t.neighbors(item):
        if n1 > item and t.degree(n1) > 1:
            st1 = st1 + neighbour(t, n1, st1, list1, list2)

    elif n1 > item and t.degree(n1) == 1:
        fruit_dictionary1 = dict(zip(list1, list2))
        ab = fruit_dictionary1[n1]
        st1 = st1 + str(ab) + "_"
        #return st1

    return st1

def prufercode_context(parent_node_list, child_node_list, explanation_node, context_node, count):
    List_of_Item = [MethodInvocation, StatementExpression, ForStatement, SwitchStatementCase, WhileStatement, IfStatement, TryStatement, ReturnStatement, LambdaExpression]

    try:
        t = ""
        tj = ""
        if (len(list1) > 1):
            list4 = []
            T = nx.Graph()
            T.add_nodes_from(parent_node_list)
            T.add_nodes_from(child_node_list)
Appendix C: Context of Code Generation

```python
edgeList = list(zip(parent_node_list, child_node_list))
edgeList1 = list(zip(parent_node_list, child_node_list, explanation_node))
edgeList2 = list(zip(parent_node_list, child_node_list, explanation_node, context_node))
T.add_edges_from(edgeList, label = 1)
T.remove_edges_from(nx.selfloop_edges(T))

nei=[]
list5=[]
list6=[]

for (node, val) in T.degree():
    if val == 1:
        list5.append(node)  # collect all terminal nodes

if T.number_of_nodes() <= 2:
k=[1,0,0,1]
elif T.number_of_nodes() > 2:
k=nx.to_prufer_sequence(T)

# print("hello0",count)

list_of_neighbour=[]
finallist_of_neighbour=[]
for eachitem in k:
    if eachitem != 0:
st12=""
    fruit_dictionary23 = dict(zip(parent_node_list, explanation_node))
    ab23=fruit_dictionary23[eachitem]
    # print(ab23)
    if ab23 in List_of_Item:
        list_of_neighbour.append(ab23+"_")
    else:
        k1=neighbourt(T,eachitem,st12,list1, list2)
        list_of_neighbour.append(k1)

elif eachitem == 0:
    list_of_neighbour.append("0"+"_")

# print(count)
# print(fruit_dictionary23)
# print(list_of_neighbour)
```
set1={}  
set2={}  

#print(list_of_neighbour)  
context_from_dictionary=""  
context_string=" "  
context_list=[]  
for obs in list_of_neighbour:  
    #print(len(list_of_neighbour))  
    set1=obs.split(".")  
    #print("hello1",count)  
    finallist_of_neighbour.append(set(set1))# after removing the duplicates  

#print(finallist_of_neighbour)  

fruit_dictionary66 = dict(zip(parent_node_list,context_node))  
fruit_dictionary77 = dict(zip(explanation_node,context_node))  
for each_neighbour_list in finallist_of_neighbour:  
    for each_item in each_neighbour_list:  
        if each_item !='':  
            if each_item in List_of_Item:  
                Context_from_dictionary=fruit_dictionary77[each_item]  
                if Context_from_dictionary != 'XXX':  
                    Context_from_dictionary=each_item+" "+Context_from_dictionary  
                    Context_from_dictionary=each_item  
            else:  
                Context_from_dictionary=fruit_dictionary66[int(each_item)]  
                try:  
                    context_string=context_string+Context_from_dictionary+" "  
                except:  
                    context_string=context_string+"nan1"+" "  
        #print(each_item,"\n",fruit_dictionary77)  
else:  
    Context_from_dictionary=fruit_dictionary66[int(each_item)]  
    try:  
        context_string=context_string+Context_from_dictionary+" "  
    except:  
        context_string=context_string+"nan1"+" "  
        #val=input("error of NaN ")
context_string = context_string + " 
pos_of_context = context_string.split()
for i in range(len(pos_of_context)):
    temp1 = pos_of_context[i]
    temp1 = ' '.join(re.split(',|\.|\ ', temp1))
    temp1 = re.split('_+', temp1)
    for j in range(len(temp1)):
        temp2 = temp1[j]
        if temp2 in context_list_constant:
            context_list.append(temp2)
        elif temp2 not in context_list_constant:
            if (temp2.isupper() != True):
                temp2 = ' '.join(re.sub(r"([A-Z])", r" \1", temp2).split())
            context_list.append(temp2)

print(len(context_list))
#val = input("next ")
context_string = ""
#val = input("enter the value ")
print("hello", count)
with open("context_of_program_for_train_TLCodeSum","a") as outfile:
    outfile.write(' '.join(map(str, context_list)) + '\n')
with open("prufer_of_program_for_train_TLCodeSum","a") as outfile:
    outfile.write(' '.join(map(str, k)) + '\n')

except Exception as e:
    #n = len(T)
    #print(count, set(T), set(range(n)))
    #print(t, "," , tj, fruit_dictionary66)
    print("ekllo", e)
    val = input("enter the value ")

import pandas as pd
list1=[]
list2=[]
list3=[]
list4=[]
mid=[]

# put this part above the Context geertation part and download the XC part
context_list_constant=[]

with open("xc","r",encoding="utf-8")as r1:
t1=r1.readlines()  #vocab.code
for i in range(len(t1)):
    context_list_constant.append(t1[i].replace('
',''))

print(context_list_constant)

# насomorphic-------------------------------------------

fd=pd.read_csv("train-pruer.csv",header=None)

pre=0
count=0

for i,j,k,l in zip(fd[0],fd[1],fd[2],fd[3]):
    if(pre <= i):
        list1.append(i)
        list2.append(j)
        list3.append(k)
        list4.append(l)
        pre=i
    elif(pre > i):
        pruercode_context(list1,list2,list3,list4,count)
        count=count+1
        list1=[]
        list2=[]
        list3=[]
        list4=[]
        list1.append(i)
        list2.append(j)
list3.append(k)

list4.append(l)

pre=0

prufercode_context(list1,list2,list3,list4,count)