CLCD-I: CROSS LANGUAGE CLONE DETECTION WITH INFERCODE

by

MOHAMMAD A A YAHYA

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE AND INFORMATICS

2023

Oakland University
Rochester, Michigan

Doctoral Advisory Committee:

Dae-Kyoo Kim, Ph.D., Chair
Lunjin Lu, Ph.D.
Hua Ming, Ph.D.
Eralda Caushaj, Ph.D.

This dissertation is dedicated to all members of my family, especially to my father, Adnan Yahya and my mother Afia.

ACKNOWLEDGMENTS

I would like to thank my family for their support. I have been sustained throughout my life by their support, encouragement, and love. Also, it is dedicated to everyone who has played a positive role in my life. Additionally, I would like to thank my supervisor Professor Kim for his patience throughout my doctoral journey. It was a pleasure working with the committee members, including Professors Lu, Ming, and Eralda. I am grateful for their positive feedback. I am also grateful for the support I received from everyone at school before meeting Professor Kim. Last but not least, I'd like to thank the dean of my school Professor Louay Chamra and Professor Qian Zou for making our school a comfortable and inspiring place to learn. I also want to thank the ISSO for helping me throughout my study to learn more about the school and the American culture.

MOHAMMAD A A YAHYA

ABSTRACT

CLCD-I: CROSS LANGUAGE CLONE DETECTION WITH INFERCODE

by

MOHAMMAD A A YAHYA

Adviser: Dae-Kyoo Kim, Ph.D.

Source code clones are common in software development as part of reuse practice. However, they are also often a source of errors compromising software maintainability. The existing work on code clone detection mainly focuses on clones in a single programming language. However, nowadays software is increasingly developed on a multilanguage platform on which code is reused across different programming languages. Detecting code clones in such a platform is challenging and has not been studied much. In this paper, we present CLCD-I, a deep neural network-based approach for detecting cross-language code clones by using InferCode which is an embedding technique for source code. The design of our model is twofold: (a) taking as input InferCode embeddings of source code in two different programming languages and (b) forwarding them to a Siamese architecture for comparative processing. We compare the performance of CLCD-I with LSTM autoencoders and the existing approaches on cross-language code clone detection. The evaluation shows the CLCD-I outperforms LSTM autoencoders by 30% on average and the existing approaches by 15% on average.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Code clones are code fragments that have the same or similar syntax and semantics [11]. They are often practiced in software development as part of reuse effort. However, code clones are in general considered hampering the quality of software due to misfit and modifications made during reuse which often introduce errors [54]. They also require that when one clone is changed, all other clones be updated for consistency [48].

Deep learning has been increasingly adopted in code clone detection [36, 79–81]. The general approach is that source code is represented in abstraction such as abstract syntax trees (ASTs), tokens, and control flow graph and embedded using embedding techniques such as Word2vec, Node2vec, and Graph2vec to learn structural similarity. Most existing work focuses on code clone detection in the same programming language [51]. However, nowadays software is increasingly developed on a multi-language platform where similar functionalities are used across multiple programming languages. For example, common APIs for big data processing such as Apache Spark have similar naming and call patterns written for different programming languages [57, 60]. Such an environment often involves clones in multiple languages and requires consistent update across clones when a change is made on one clone. There exists some work [57, 60] on cross-language clone detection. However, their models suffer from poor performance mainly due to low quality features for training. This work presents CLCD-I, a cross-language clone detection approach using InferCode [16], which is an embedding technique for source code. The model takes as input the pre-trained embeddings from InferCode to learn abstract features of source code by grouping related embeddings. The embeddings are fed into a Siamese architecture [15] for comparative processing of two different languages. In this work, we focus on clones of III in Java and Python code. This work is evaluated the model using a dataset containing more than 40,000 pairs of Java

and Python code snippets. The evaluation shows a 10% improved accuracy over the existing work.

1.1 Challenge in Source Code Clone Detection using Deep Learning

There are two main challenges associated with any source code problem. Firstly, how can a program be decomposed into smaller building blocks constrained by two criteria: meaningful and small enough to repeat. Second is how to combine these parts together again to get a single prediction. A small building block might be a path in an AST, a subtree in an AST, a token in an AST, or a token in source code. However, not every building block can be used for all downstream tasks in source code, such as detecting code clones. Source code tokens from an AST or from raw source code may work for code clones of Type I and II, but may be more challenging for Type III and Type IV.

For the first challenge of decomposing a program, the less effort invested in feature selection the longer it takes for the deep learning model to make sense of data to learn semantic and syntactic information. If more hand crafted features are focused as in [57] the more language and task specific the features become. Another example would be the AST path, which is purely syntactic and thus can't enable semantics to be leveraged when fed to a model. As a result, it can be used for many different programming languages and tasks. [5]. Fig. 1.1 represents an example of AST path taken between two leaves in Alon et al. work. Despite that AST path is purely syntactic, if you read it from left to right as in Fig. 1.1, you will get some semantics.

1.2 Motivation

Due to the increasing amount of source code, large-scale clone detection is becoming increasingly necessary. Several new applications of clone detection have emerged as a result of large code bases and project repositories, including mining library candidates [37], detecting similar mobile applications [19], and searching for code [42, 43]. Although these modern

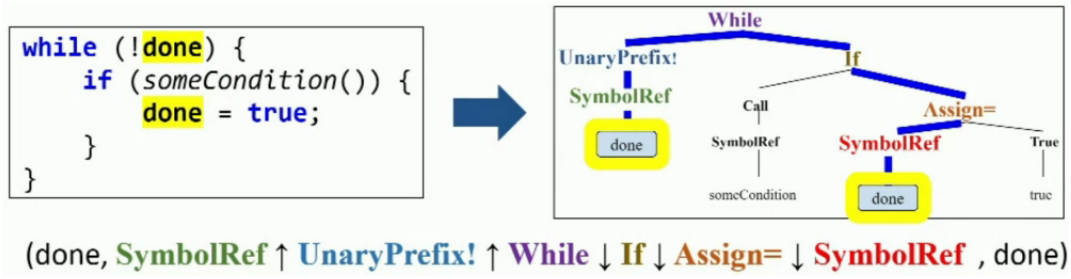


Figure 1.1: AST path taken between *done* and *done* leafs [5].

use cases present new opportunities for clone detection, they also pose challenges regarding scalability.

An example of the problem's size and scale can be illustrated with a real-life scenario. An e-commerce website and database were built for a company by a recruiting company. Except for the CSS, the website has the same functionality. Each bank had a separate code base that was maintained by the team for the website deployed for many different clients. Eventually, they decided to create a common code base for all these banks to minimize expenses incurred by: (1) re-creating common features, and (2) dividing up existing code bases into common parts to maintain those parts exclusively.

Due to the large number of code lines in the website codebase, finding common functionalities is the first step in separating them. Sourcerr team confirms that it becomes even harder as the codebase grows, which impacts the scalability of the e-commerce website. They tried two different tools, CloneDR and Simian, and neither scaled [65].

1.3 Research Questions

The performance of CLCI-I is compared with LSTM autoencoders, which are widely used for the translation of languages. Furthermore, CLCD-I is compared with the existing approaches on cross-language code clone detection. Specifically, the following research questions are sought.

- Studying the challenges associated with detecting clones in cross-language setting. Why it's easy to detect some common features of languages while it's hard to detect other features?
- Is CLCD-I more effective than LSTM autorencoders?
- Testing multiple variants of CLCD-I by varying the network architecture depth and width?
- Is CLCD-I more effective than existing approaches?
- Can cross language clone detection be applied?
- Does deep learning model scaling helps to learn more about InferCode embeddings?

1.4 Contribution

The following contributions are made as part of the investigation of the above research questions:

- It's found out that CLCD-I outperforms all existing cross language clone detection tools by 20%.
- The CLCD-I powered by InferCode, which is a self-supervised deep learning model, confirms that unsupervised self-learning neural models can learn more features from source code that could help detect clones.
- An assessment of CLCD-I's potential for use in large-scale software was conducted and a solution was proposed.
- How model scaling is useful to learn more features from InferCode embeddings by increasing the width and height of the network is investigated.

CHAPTER 2

BACKGROUND

It may not be a good practice to clone code because it can increase maintenance costs and degrade software quality, but it can also be useful for code simplification, code maintainability, plagiarism detection, copyright infringement detection, malicious software detection, and bug report detection [83].

2.1 Code Clone Detection Types

The following formal definitions of source code clones are used as a base in this work:

- **Type 1 clones:** Assume that C represents a collection of code segments, and consider $f_1, f_2, ..., f_n$ as unique entities in C. A code snippet f_i is denoted as a Type 1 clone of another code snippet f_j if, and only if, both f_i and f_j maintain syntactic identity, inclusive of both comments and whitespace.

- **Type 2 clones:** Let C signify a pool of code fragments, and $f_1, f_2, ..., f_n$ be distinct constituents of C. A code segment f_i qualifies as a Type 2 clone of a different code segment f_j if there exists a series of syntactic transformations, such as the renaming of variables, that can morph f_i into f_j.

- **Type 3 clones:** Suppose that C constitutes a group of code fragments, and $f_1, f_2, ..., f_n$ serve as discrete elements of C. A code fragment f_i is deemed a Type 3 clone of another code fragment f_j if and only if f_i and f_j achieve similar functions but differ in the specifics of their implementation.

- **Type 4 clones:** Assume C stands for a set of code segments, and $f_1, f_2, ..., f_n$ act as individual members of C. A code snippet f_i is labeled as a Type 4 clone of a separate code snippet f_j if and only if f_i and f_j exhibit either syntactic or semantic similarity,

yet they deviate in the functions they perform. Further analysis might be necessary to discern whether they represent actual clones or merely incidental similarities.

Code cloning detection involves comparing two or more fragments of code with respect to a clone type [69]. Code clone pair is two code fragments that are either similar or identical fragments. Generally, code clones can be classified into four types [12, 63].

Fragments of Type 1 in Fig. 2.1 that are identical except for comments and layout variations [3, 12]. It can be represented by the following function as a simplified representation of clone detection:

$$\text{isClone\_TypeI}(codeFragmentA, codeFragmentB) \tag{2.1}$$

CodeFragmentA and codeFragmentB would be considered Type I clones if they are the same, and false otherwise. As a result of stripping out all whitespace, layout, and comments from the code fragments, the function compares the two code fragments. A pseudo-algorithm to detect Type I clones might look like this in the Listing 2.1.

```
1  function isClone_TypeI(codeFragmentA, codeFragmentB) {
2  //Normalize the code fragments by removing whitespaces, layout,
     comments
3  normalizedCodeFragmentA = normalize(codeFragmentA)
4  normalizedCodeFragmentB = normalize(codeFragmentB)
5
6  // Compare the normalized code fragments
7  if (normalizedCodeFragmentA == normalizedCodeFragmentB) {
8    return true
9  } else {
10   return false
11 }
12 }
```

```
13
14 function normalize(codeFragment) {
15   // Remove whitespace
16   codeFragment = removeWhitespace(codeFragment)
17
18   // Remove layout (like line breaks, tabs, etc.)
19   codeFragment = removeLayout(codeFragment)
20
21   // Remove comments
22   codeFragment = removeComments(codeFragment)
23
24   return codeFragment
25 }
```

Listing 2.1: Pseudo algorithm to detect Type I clones.

```
1 public class MyClass {
2     public static void main(String args[]) {
3       int x=10;
4       int y=x+2;
5     }
6 }
```

```
1 public class MyClass {
2     public static void main(String args[]) {
3       int x=  10; //spaces added
4       int y=x+2;
5     }
6 }
```

Figure 2.1: Type 1 Example Clone-pair.

A Type 2 fragment in Fig. 2.2 is identical to a Type 1 fragment, except for variations in the identifier name and literal [3, 12]. Fundamentally, Type I clones represent verbatim duplications, whereas Type II clones exhibit structural congruity yet differ in discrete elements such as identifiers, literal values, and types. These varieties of clones might illuminate segments of code which offer opportunities for refactoring, thereby enhancing maintainability

and diminishing the propensity for bugs. It is noteworthy that certain clones emerge from requisite duplication or incidental resemblance, and thus, not all clones should be deemed detrimental. Often, the determination of the problematic nature of a detected clone hinges on the contextual circumstances and the particular requirements of the project at hand. A pseudo-algorithm might look as in Listing 2.2.

```
1   //Function to check if two code fragments are Type II clones.
2 public boolean isClone_TypeII(String fragmentA, String fragmentB)
      {
3   // Normalize the code fragments
4   String normalizedFragmentA = normalize(fragmentA);
5   String normalizedFragmentB = normalize(fragmentB);
6
7   // Compare the normalized fragments and return the result
8   return normalizedFragmentA.equals(normalizedFragmentB);
9 }
10
11 //Function to normalize a code fragment. Replaces all identifiers
     , literals, and types with a common symbol and removes all
     whitespace, layout elements, and comments.
12
13 public String normalize(String fragment) {
14   // Normalize the fragment
15   fragment = replaceIdentifiers(fragment);
16   fragment = replaceLiterals(fragment);
17   fragment = replaceTypes(fragment);
18   fragment = removeWhitespace(fragment);
19   fragment = removeLayout(fragment);
20   fragment = removeComments(fragment);
21
```

```
22    // Return the normalized fragment
23    return fragment;
24 }
```

Listing 2.2: Pseudo algorithm to detect Type II clones.

Basically, the output of the code above is a normalized code with common symbols. Common symbols is a placeholder for all code identifiers, literals, constants and types. By following this, code fragments will be compared purely based on their structure. Below is an example of code before and after normalization in Listing 2.3.

```
1 //Before normalization
2
3 //FragmentA
4 int x = 10;
5 x = x + 5;
6
7 //FragmentB
8 float y = 20;
9 y = y + 10;
```

Listing 2.3: Two code fragments to normalize.

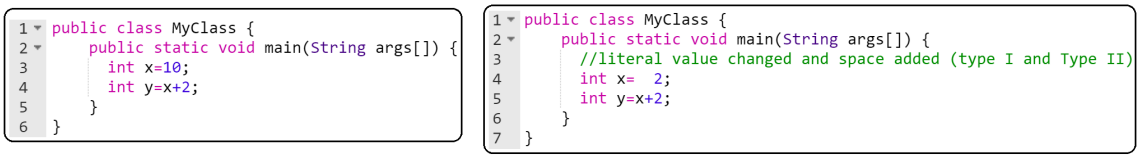
After passing fragments to isClone_TypeII, if identifiers (x and y) are replaced with a common symbol ID, literals (10, 5, 20, 10) with LIT, and types $(int, float)$ with TYPE we get in Listing 2.4.

```
1 //After normalization
2
3 //FragmentA
4 TYPE ID = LIT;
5 ID = ID + LIT;
6
```

```
//FragmentB
TYPE ID = LIT;
ID = ID + LIT;
```

Listing 2.4: After normalization.

In spite of the differences in variable names, types, and values, these normalized fragments are identical, indicating that the original fragments are clones of each other in Type II.



```
public class MyClass {
    public static void main(String args[]) {
      int x=10;
      int y=x+2;
    }
}
```

```
public class MyClass {
    public static void main(String args[]) {
      //literal value changed and space added (type I and Type II)
      int x=  2;
      int y=x+2;
    }
}
```

Figure 2.2: Type 2 Example Clone-pair for adding 2 integers.

Type III is much more complex than Type I and Type II. Differences in syntactic structure are classified as Type 3 [3, 12], where statements are added, modified or removed from one another in addition to clone Type 2. Type 3 in Fig. 2.3 has Type 1 as well due to spaces between variables x and y. Simply put, cloned source code fragments of Type III are syntactically similar despite adding, modifying, or removing statements. In most cases, the similarity score between 2 code fragments is determined by the number of statements that match between the two fragments. The similarity score is defined S between code fragments A and B, defined as:

$$S(A, B) = 2 \times M/(A + B) \tag{2.2}$$

where M being the number of matching statements in A and B, and $|A|$ and $|B|$ being the total number of statements in each code fragment. To illustrate that with an example, the term $2 * M$ in Eq. 2.4 to account for the fact that matches from both code fragments are considered, A and B. Each matching statement is a part of both A and B. Suppose A has 5 statements, B

has 7 statements, and there are 3 matching statements. Then, M equals 3, $|A|$ equals 5, and $|B|$ equals 7. The score $S(A, B)$ is then calculated as $2 * 3/(5 + 7) = 6/12 = 0.5$. This means that half of the statements in A and B are matching. If M was not multiplied by 2, the score would be 0.25, which would not accurately reflect the proportion of matching statements in the two code fragments. Therefore, by multiplying both code fragments by 2, the similarity score reflects the proportion of matching statements in both. If $S(A, B)$ is greater than or equal to a certain threshold (commonly 0.7 or 70%), A and B are considered as Type III clones of each other. Clone thresholds are typically derived from empirical studies, and

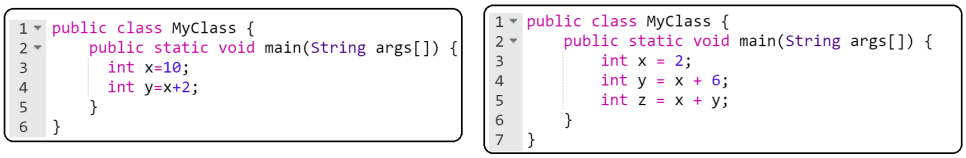


Figure 2.3: Type 3 Example Clone-pair for swapping 2 integers without a temporary variable.

no single value fits all. For Type III clones, a threshold of 0.7 (or 70%) is commonly used because it offers a reasonable balance between precision and recall[62]. Too high of a threshold might miss out on potential clones that aren't exactly the same but are very similar (high precision, low recall). On the other hand, if the threshold is set too low (close to 0), different code fragments might be incorrectly classified as clones (low precision, high recall). These concerns are generally balanced by the 70% threshold. This is what we set our Siamese network to.

Type 4 fragments are syntactically dissimilar, yet it implements the same functionality [3, 12]. This is the most sophisticated and the hardest to find. The two methods in Fig. 2.4 are supposed to swap 2 integers. While first method on the left does swapping operation using temporary variable $temp$, the second one does the same thing without any temporary vari-

able. So it can be noted that there are no addition of statements or deletion of statements here but clearly a different application apparently but internally it does the same functionality. The process for detecting code clones consists of two phases: first, the code is transformed

```java
public class MyClass {
    public static void main(String args[]) {
        int temp;
        int x = 2;
        int y = 3;
        temp = x;
        x = y;
        y = x;
    }
}
```

```java
public class MyClass {
    public static void main(String args[]) {
        int x = 2;
        int y = 3;
        x = x + y;
        y = x - y;
        x = x - y;
    }
}
```

Figure 2.4: Type 4 Example Clone-pair for swapping 2 integers without a temporary variable.

into another representation, followed by the actual comparison. The clone detection algorithms can be divided into four main categories: text-based techniques, token-based techniques [10, 57], tree-based techniques [60], graph-based techniques [53]. Techniques under text-based approach uses the raw code and compare clones based on string matching. A token-based model converts the code into tokens and then finds duplicate sub-sequences of tokens, but it is prone to false positives [62].

In a tree-based model, tree matching can be performed on either parse trees or abstract syntax trees (ASTs), which contain syntactic information derived from source code. By contrast, graph-based techniques rely on static code analysis to extract semantic information. Before finding semantic similarities, the program is first converted into a program dependency graph (PDG) [47] or control flow graph (CFG) [29]. Statements or expressions are represented by nodes, while data dependencies are represented by lines.

2.1.1 Syntactic and Structural Concepts in Code and Natural Language

Type III clones are more about structural similarities between 2 code fragments or more. It's important to understand these concepts prior to understanding Type III clones between 2 code fragments. In essence, both syntactic and structural relations relate to the organization of elements, but they refer to different types of data (natural language versus code), so their implications differ. In order to distinguish these differences, the terms "syntactic" and "structural" are used.

Syntactic properties of code refer to the rules for programming in a specific language [2, 67, 68]. There are many aspects to this, including the correct placement of braces and semicolons, the structure of commands and expressions, etc. In Python, blocks of code are defined by their indentation level, which is a basic syntactic rule. Parsing, or syntactic analysis, examines whether code follows a programming language's grammar. In contrast, structural properties of code refer to the high-level organization of code and its constructs [2, 67, 68]. In object-oriented programming, it includes aspects such as control flow (loops, conditionals, function calls), data flow (where and how data moves), and class organization. Code structural properties examine how the program's components fit together, in a sense examining its architecture. Although these properties are related, they focus on different aspects of code. The syntax analysis focuses on the correctness of the code according to the rules of the language, while the structural analysis focuses on the design and organization of the code. Understanding and analyzing code requires both.

In NLP, however, since InferCode is built on concepts similar to those used in NLP, we also have the same terms structure and syntax even though they mean differently. In NLP, syntax is the set of rules, principles, and processes that govern the structure of sentences [21]. Syntax in linguistics refers to word order, sentence structure, and grammatical rules [21]. The process involves tagging parts of speech (nouns, verbs, adjectives, etc.), parsing sentences into trees to identify grammatical relationships between words, and applying grammatical transformations. The term structure is not commonly used in NLP, but it refers to

higher-level, more abstract aspects of language beyond individual sentences. Among them are discourse analysis (understanding the logical sequence of sentences), narrative analysis (studying how stories are constructed), and pragmatics (understanding how context influences interpretation).

The selection of AST to represent the structure of code stems from many reasons. Hierarchical structure is captured by ASTs: ASTs explicitly encode parent-child relationships between different elements of the code. As a result, the source code's structural information is preserved, which is essential for understanding how the code functions. This is particularly important in tasks such as Type III clone detection, which aims to find code fragments with similar semantics but different textual properties. In ASTs, unnecessary details are abstracted away. Parentheses and certain formatting are discarded by ASTs because they are irrelevant to the meaning of the code. As a result, the syntactical structure and semantics of the code can be more focused, which is crucial for tasks like Type III clone detection. The tree structure of an AST facilitates recursive algorithms, which makes it easier to write algorithms that operate on the code. There are many types of calculations, including traversals, transformations, and metrics calculations.

To summarize, the AST inherently is constructed based on the grammar of a specific programming language. AST captures both aspects of code the structure and syntax. In terms of syntax, an AST is constructed according to the rules of the programming language. In the tree, each node represents a construct in the source code. A node can be an expression, a statement, a declaration, or even a block of code. By arranging and connecting these nodes in a way that mirrors the code's structure in accordance with the language's grammar, ASTs implicitly store syntactic information. In terms of structural aspect, an AST reflects the hierarchical organization of source code with its tree-like structure. In a function call node, the name of the function and its parameters might be represented as child nodes. Parent-child relationships are established by the edges of the AST, which show how individual code parts are grouped and nested. When interpreting code, this structural aspect is crucial, since the

same set of operations can have different outcomes depending on how they are ordered and grouped.

2.1.2 Type III Clones Based on AST

It's important to note that Type III source code clones are not about the exact match, but about code that does similar functionality to another with further modifications such as changed, added, or removed statements. So, due to modifications, these clones may not be syntactically identical, but they perform the same computation or function [40]. However, Type II seek after exact match. Type II clones are syntactically identical fragments except for differences in identifiers, literals, types, whitespace, layout, and comments. Normalizing the code fragments by renaming variables or replacing literals, will result in exactly identical fragments [38]. Therefore, clones of type III perform the same function but differ in structure. As a consequence, their detection requires more than a simple textual comparison or token level analysis.

An AST is a tree representation of the syntactic structure of source code. Each node of the tree denotes a construct in the code. The root of the tree usually represents the entire program, while the leaves represent low-level constructs such as identifiers or literals.

The differences between Type III clones based on their respective ASTs are illustrated as follows. Here's a simple example of AST for the code segment `x = a + b` and `x = a - b` as we see in Fig. 2.5. In order to illustrate how Type III clone detection works using ASTs, let's examine two pieces of code fragments that are similar but not identical. ASTs represent the structure of code fragments. In this way, structural similarities and differences are easier to detect, even if the actual statements or expressions differ. One way to find Type III clones is based on tree-based similarity measures, such as subtree isomorphism, which checks whether two trees have similar (isomorphic) subtrees. Nevertheless, Type III clone detection involves detecting similar code fragments, which have been modified in some manner. Therefore, it is not sufficient to simply look for identical subtrees. In this regard, more flexible mea-

sures of similarity, such as tree edit distance, are more suitable for finding Type III clones. In

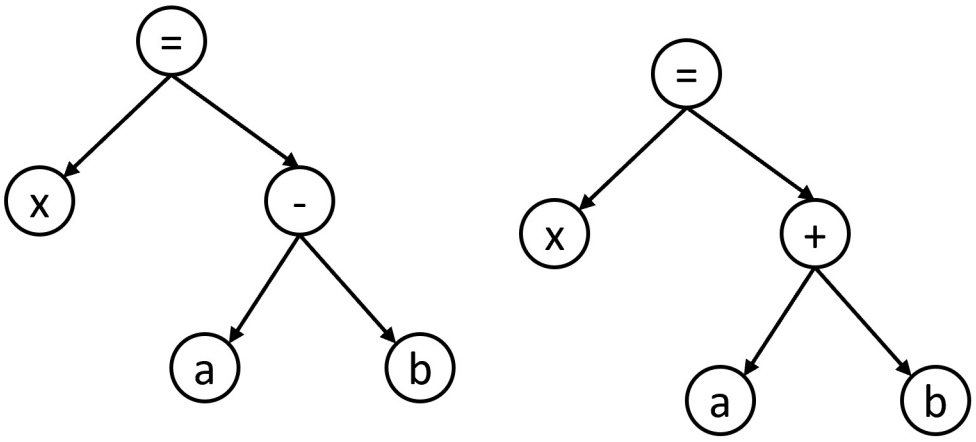


Figure 2.5: On the right we have AST tree for`x = a + b` and AST tree for `x = a - b` on the left.

an edit distance calculation, one tree is transformed into another by a sequence of operations (insertions, deletions, renamings). To make two trees identical, the cost is the total number of operations required. We can walk through the calculations of how we can compute Type III clones based on ASTs for code fragments in Fig. 2.5. Commencing with the root nodes of both Abstract Syntax Trees (ASTs), we encounter no divergence, as both roots are "=", rendering no modification cost. As we proceed to traverse the trees in parallel, we note that the left child of both roots, "x", remains consistent across the trees, once again eliminating the need for any operational adjustments. However, the harmonious traversal begins to encounter disparities at the right children of the root nodes: while in AST1 the right child node is "+", AST2 presents a "-", requiring us to undertake a renaming operation, whether it be "+" to "-" or vice versa, accruing a cost of 1. Finally, turning our attention to the children of the nodes "+" and "-", we find a return to consistency, as they manifest as "a" and "b" in both trees, ob-

viating the need for any further operations. Thus, through this meticulously navigated journey through the nodes of the ASTs, we can calculate the presence of Type III clones.

Above, edit distance with cost 1 is considered as acceptable threshold for Type III clones, but to formalize the discussion, researchers adopt relative threshold with regard to the size of code fragment [39]. Depending on the specific context, size of the code fragments, and the particularities of the software system under study, the threshold for tree edit distance can vary from two code fragments to be considered Type III clones. Clone detection researchers or developers typically set thresholds on a case-by-case basis. However, several academic studies suggest that a relative threshold may be more appropriate than a fixed one. Instead of setting a fixed number of operations, the threshold is based on the size of the code fragments. Relative thresholds have the advantage of accounting for the fact that larger code fragments often require more edit operations to transform from one to the other, even if they have the same function. This means that if the tree edit distance is less than or equal to 30% of the size of the code fragments, they are considered to be Type III clones. A distance of 1 would be considered similar if we set the threshold at 30%, and considering the size of our code fragments (4 nodes).

2.1.3 Type III Clones Based on InferCode

It is possible to embed source code snippets into a high-dimensional space using techniques such as source code embedding. In this process, the code is parsed into an AST, the nodes are walked through in a certain order (such as depth-first or breadth-first), and the types of nodes and their relationships are encoded in a vector of the code. The snippets of code that are similar in size should be near each other in a vector space.

InferCode uses convolutional filter that is mainly used in CNN. A CNN is primarily used in tasks with grid-like topologies, such as images (2D grids of pixels) and time-series data (1D grids over time). The strength of using convolutional filter is that we are able to capture local and spatially-invariant features. Thus, we can identify features (such as edges and

textures in images, or local syntactic patterns in ASTs) regardless of where they appear in the input. The convolution operation in a CNN applies the same filters across the entire input, allowing it to detect the same feature anywhere in the input. This property is known as translation invariance.

InferCode is trained to predict subtrees of its AST. InferCode learns code representations by predicting subtrees of the AST of a code snippet. Here is how it works:

For each node in an AST, a subtree is extracted with that node as the root. The size of the subtree (i.e., the number of descendant nodes included) is controlled by a parameter. For each extracted subtree, a negative sample is generated by randomly selecting another node from the same AST and swapping it with the root. As a result, a "corrupted" subtree is created, which is unlikely to appear in the original code. A tree-based neural network model encodes both the original and corrupted subtrees into vectors. Finally, for the loss function, a learning objective is to make the original subtree encoded closer to the entire code snippet encoded to an encoding that is closer to the corrupted subtree encoded in the vector space. With its ability to predict subtrees of the AST, InferCode is able to capture both the local and global structure of the code, as well as the semantics of each element.

For learning code representations, InferCode uses a self-supervised learning approach. By generating "virtual" labels from existing, unlabeled data, self-supervised learning reformulates unsupervised learning problems into supervised ones. This is known as a pretext task. The first step is to define a pretext task that can be used to guide the learning process. In InferCode, the chosen pretext task is predicting a subtree of the AST given the rest of the AST as context. Next is virtual label generation. The "virtual" labels are the subtrees of the AST, chosen based on some heuristic or selection algorithm. For example, you could select all subtrees rooted at a certain depth in the AST. This is a flexible step, and different criteria for selecting subtrees could lead to different learning representations. Once the labeles are ready, the loss function is needed to train on virtual labels. With labels, a supervised loss function can be defined based on the pretext task. The aim is to train a model that minimizes

this loss function. In InferCode, the loss function could be defined as a distance or dissimilarity measure between the context and subtree embeddings. For instance, it could be the negative cosine similarity or the Euclidean distance. During training, the model produces intermediate representations of the data that correspond to the virtual labels while minimizing the loss function. These are the embeddings of the subtrees and the context in InferCode. Because these representations are learned by predicting meaningful parts of the AST, they are expected to carry good information about the structure and semantics of the code (the subtrees). The intermediate representations can be used for a variety of downstream tasks once they are learned. In addition to being versatile and robust, the learned representations are not tied to a specific task. Unlike some existing works using self-supervised learning to represent code, this approach is not task-specific. Conversely, InferCode uses more general representations that can be applied to a wide range of tasks.

In the training phase, the process commences with the random initialization of parameters in the Tree-Based Convolutional Neural Network (TBCNN). This network serves the purpose of converting the ASTs into vector embeddings. During the training for each code fragment, InferCode designates a randomly selected subtree as a "virtual" label. This subtree and the remaining part of the AST (referred to as the context) undergo encoding into vector embeddings via the TBCNN. The encoding operation comprises a convolutional process over the AST nodes, pooling to encapsulate the most significant features, and fully connected layers to generate the ultimate embeddings. The subsequent phase involves the computation of the loss function, premised on the disparity between the context's embeddings and the subtree. Depending on the metrics applied, such as cosine similarity or Euclidean distance, the goal could be to either maximize the similarity or minimize the distance. Using backpropagation, gradients of the loss function relative to the parameters of the TBCNN are computed. The parameter update process then follows, usually with the application of gradient descent or its variants, such as Adam or RMSprop. In summary, the overarching aim of the training process is to fine-tune the parameters of the TBCNN. This refinement is geared towards en-

abling the model to generate an embedding, given the context, which closely aligns with the embedding of the corresponding subtree. Such a mechanism encourages the model to learn representations encapsulating the structural and semantic information embedded in the code.

Similarity is defined and measured differently in this source code embedding-based approach than in tree edit distance-based methods. Similarity between trees is measured as the minimum number of operations needed to transform one tree into another, with each operation costing a certain amount. ASTs are compared with fine-grained similarity measures that take their exact structure into account. Embedding-based methods, on the other hand, measure similarity in terms of the distance between vectors. With this method, we can capture semantic relationships among code snippets, rather than just their exact structure, as a measure of similarity. In the embedding space, two code snippets performing the same calculation differently might be far apart in terms of tree edit distance, but close in terms of tree edit distance.

To understand why InferCode approach is better suited for capturing semantics than tree edit approach, we should look deeper into the nature of both approaches. InferCode uses an embedding approach that considers not only the individual nodes of the AST, but also their relationships to one another, as well as the overall structure of the code. Through these embeddings, code snippets are mapped into a high-dimensional space where the 'distance' between any two points reflects their semantic similarity. Often, this is achieved through the architecture of neural networks. A Recursive Neural Network (RNN), for instance, would combine the representations of the child nodes to form the representation of the parent node. As a result, the representation of a node in the AST depends not just on the node itself, but also on its context within the code. By training the neural network to create representations of code snippets that preserve semantic relationships, the semantics are captured. In order to learn these semantic relationships, a suitable loss function is often applied during training. With RNNs, the AST is processed bottom-up by combining the representations of the children nodes to form the parent node representation. The key property here is that the way

information is combined is learned from the data, and it can capture complex relationships between the nodes. Often, this can be accomplished through a gating mechanism, such as those used in Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) units. Imagine an AST with a root node representing a “for” loop and two child nodes representing “initialization” and “increment”. In order to form the representation for the “for” loop, the RNN would first learn representations for “initialization” and “increment” operations. As a result, “for” loops are represented based on their context in code, capturing their semantics.

The “spatial” relationship between an AST and source code can be viewed as its syntactical arrangement. In a CNN used by InferCode, each filter can recognize specific local syntactic patterns, and the convolution operation ensures that these patterns can be recognized anywhere in the AST. A CNN can be applied hierarchically so that lower layers capture small local patterns (like specific programming constructs or variables) and higher layers combine them to represent larger code segments. Thus, CNNs capture hierarchical syntactic patterns from ASTs. In this way, a broader range of syntactic properties can be captured in the source code. As a result of the way the AST is processed by InferCode compared to tree-edit distance, it is considered to capture semantics more effectively than a tree-edit distance approach. In contrast to its semantic capability, the tree-edit distance approach is a comparison operation based on transformations (insertion, deletion, and substitution), which is more about the structural similarity of the two ASTs. A CNN-based approach in InferCode, on the other hand, is capable of recognizing patterns in the code structure directly, thus capturing its semantics more directly.

Consequently, CNNs and RNNs capture semantics in code through the recognition and representation of meaningful patterns in the AST. RNNs combine information from child nodes to represent parent nodes and propagate information through their hidden states. CNNs identify local patterns and preserve their spatial relationships while CNNs identify local patterns and observe their spatial relationships. By doing so, they are able to capture both local and global semantic relationships, as well as short-range and long-range relationships.

Since this work depends on how InferCode defined the pretex task, the pretex task used in InferCode is mathematically defined, which is not elaborated clearly. InferCode's pretext task is based on self-supervised learning and can be formulated as follows. Suppose we have a dataset D of code fragments. AST(c) represents each code fragment c in D as an abstract syntax tree (AST). From each AST(c), InferCode randomly picks a subtree s and considers it a "virtual" label. In context(c), s, AST(c) and AST(c) are the subtrees and the remaining part of AST(c). A labeled pair (context(c), s) is generated for each code fragment c in D, where s represents a randomly selected subtree from AST(c). The training data for the model is this labeled pair. InferCode's pretext task consists in learning a function f (parameterized by the TBCNN weights) that maps context(c) to an embedding E(context(c)) and s to an embedding E(s), thereby ensuring that their embeddings are similar. Mathematically, the pretext task aims to minimize the distance between E(context(c)) and E(s) for all pairs of (context(c), s). Cosine similarity or Euclidean distance can be used to measure distance. Using d as the measure of distance, the objective function is as follows:

$$L = \sum_{c \in D} \text{Distance}(\text{E(context(c))} - \text{E(s)}) \tag{2.3}$$

where L represents the total loss for the training data. Gradient descent or its variant is used to optimize this function.

The major part of InferCode is TBCNN that encodes both the randomly selected subtree and code fragment including the selected subtree as the virtual label. TBCNNs are Convolutional Neural Networks (CNNs) adapted to process tree-like structures, such as Abstract Syntax Trees (ASTs). CNNs are typically designed for grid-like data (e.g., images). TA TBCNN architecture includes a convolutional layer that processes AST nodes, a dynamic pooling layer that aggregates output from the convolutional layer, and a fully-connected layer that generates final embeddings. Next, how TBCNN works is described. It begins by embedding AST nodes, where each node is assigned an initial vector representation. This can be based on the node type (e.g., "for", "if"), and it can be learned as part of the model training.

Similar to Word2Vec, the process involves representing words as dense vectors, as in natural language processing. Then, convolution filters are applied, where a set of filters (also vectors, and also learned during training) are convolved over the AST nodes. Each filter is applied to a node and its immediate neighbors in the tree (i.e., its children), resulting in a scalar value. For each node, this step produces a vector containing the results of applying each filter. Once the outputs from the nodes are ready after applying convolution, the outputs of the convolution for all nodes in the tree are aggregated using a pooling operation, usually max pooling. This is a useful method to produce a fixed-length vector, regardless of the size or shape of the original AST. To generate the final embedding, one or more fully-connected layers can use the pooled vector. During training, these layers' weights are learned.

InferCode performs TBCNN process twice for each piece of code: once for the whole AST as the context (c), and once for the selected subtree (s). In the training process, the parameters of the TBCNN (including the node embeddings and convolution filters) are adjusted in such a way that the embeddings of the context and subtree are as similar as possible (e.g., cosine similarity, Euclidean distance) based on a given measure of similarity. Consider TBCNN$(\cdot)$ as the TBCNN operation, E as the node embedding matrix, c as the context, and s as the subtree. Then, the training of InferCode can be expressed mathematically as minimizing the loss function:

$$L(E, c, s) = \text{distance}(\text{TBCNN}(E, c), \text{TBCNN}(E, s)) \tag{2.4}$$

where distance$(\cdot)$is a function that measures the dissimilarity between two vectors. The training process adjusts the parameters (including E) to minimize this loss function.

How TBCNN works is compared with regards to Word2Vec used in natural language processing. In natural language processing, word embeddings are used to represent words as high-dimensional vectors (i.e., points in a high-dimensional space). This method aims to capture the semantic meaning of words and their relationships with one another. The embeddings are learned so that words appearing in similar contexts have similar embeddings.

The Word2Vec method is commonly used for learning word embeddings. In this system, two models are trained on large amounts of text data: Continuous Bag-of-Words (CBOW) and Skip-gram. The purpose of both models is to infer a word's meaning based on nearby words. With the CBOW model, it is possible to predict target words (such as 'cat') from source context words (such as 'the', 'sat', 'on', 'the', 'mat'). In contrast, Skip-gram predicts target words based on source context. It learns to represent semantically related words with similar vectors by training on pairs of words that occur close together in the text data. Both Word2Vec word embeddings and InferCode's TBCNNs aim to learn vector representations that capture meaningful characteristics of the data. Word2Vec captures semantic and syntactic similarities between words, while TBCNN captures structural and semantic similarities between AST nodes. A loss function is minimized by training the model on large amounts of data, and adjusting embeddings to minimize it.

2.1.4 Traditional Single versus Cross Language Source Code Clone Detection

In software systems written in a single programming language, single language source code clone detection involves identifying similar or duplicate code segments. It is possible that developers copy and paste similar code segments, reuse old codes, or write similar logic independently. The clone detection process can be used for a variety of purposes, including finding code that needs refactoring, identifying inefficient or poorly written code, detecting license violations in open-source software, and finding bugs. [9, 45–47, 62, 63].

It is important to keep in mind that the process of finding clones in Python and Java differs from the process of finding clones within a single language. There are a number of differences between programming languages, mostly because each has its own unique syntax and features [45, 62, 65, 76].

This work studies the differences between Python and Java since only considered these languages are considered. Starting with Python, syntax and semantics of programming languages differ. The Python programming language is an interpreted high-level language

with dynamic typing and a mix of imperative, object-oriented, and functional programming paradigms. Codes written in this language are characterized by readability and simplicity, which emphasize code readability. Code blocks are often indicated by indentation in Python. In contrast, Java is a statically-typed, object-oriented language. Coding blocks are denoted with curly braces and have stricter syntax. The structure, keywords, and syntactic elements of the language are therefore taken into account by clone detection tools when identifying similar code. Tools can be customized to detect clones according to a language's syntax and coding patterns when detecting clones within one language. It is crucial that the tool can accommodate the differences between languages when detecting clones across multiple languages.

Similarity measurement can be viewed mathematically as the problem of clone detection. A code fragment can be represented as $f(code)$. This work finds code fragments A and B where the similarity measure $S(f(A), f(B))$ is higher than a certain value. It is possible to have different definitions of f and S. S could be a function that computes the cosine similarity between vectors or the tree edit distance between vectors, for example. For multi-language clone detection, it is important to define an appropriate f and S that takes into account differences in syntax and semantics. There is no definitive solution that works well in every case yet for multi-language clone detection.

The abstract syntactic structure of code is represented by an AST tree. In AST-based clone detection algorithms, source code is transformed into ASTs and similar subtrees are found. Language grammar heavily influences the construction of an AST. Although the logic in the Python and Java "if" statements is the same, the ASTs will differ. This means that clone detection that works across different languages must take these differences into account.

An abstract syntactic tree is a representation of the abstract syntactic structure of a source code, which can be seen as a tree of that source code. It is built according to the grammar of the programming language the source code is written in. The aim of this work when using ASTs for clone detection is to find subtrees representing chunks of source code that

are similar, representing chunks of coding that perform similar functions within these ASTs. Let's compare Python and Java programs containing the same "if" statement logic. In spite of the same logic, Python and Java have different syntaxes and grammars, so the structure of the code will be different. Under the "if" statement, Python uses indentation, while Java uses braces "{}". As a result of this difference in syntax, different ASTs are generated for the same logical structure.

An example of detecting cross language clone between Python and Java traditionally without using deep learning approach is considered. First the code in Python for "if" statement. The AST for this Python code will have a structure representing the "if" statement, the condition, and the indented block of code.

```
1 if condition:
2   do_something()
```

While logically equivalent, the AST for this Java code will have a different structure, including braces to identify the block of code that should be executed if the condition is met.

```
1 if (condition) {
2   doSomething();
3 }
```

Therefore, if we clones between Python and Java code are detected without deep learning, an AST-based method may not be enough since the ASTs can look quite different even for the same logic. In order to account for these differences, additional logic or steps need to be added to the clone detection process, such as normalizing or standardizing the ASTs before comparing them. Detection of clones across multiple languages is, therefore, more complex than detecting them across single languages.

The following is the ASTs for the code snippets above in Fig. 2.6. This is a simplified view of the AST. The actual AST would also include the nodes and subtrees for the method calls, variable identifiers. First, source code is parsed into their corresponding ASTs. A variety of existing libraries and tools can be used to accomplish this. For Java, `ANTLR` or java-

parser might be used, and for Python, the built-in `ast` module might be used. Python and Java have very different AST structures, so a way to compare them agnostically is needed. An approach that could be adopted would be to abstract away language-specific syntax and retain only semantic information in the AST. A set of language-agnostic abstract nodes or concepts could be used to map specific AST nodes. As an example, representing any "if" statement can be considered, regardless of language, as a general "conditional" node. Then, the ASTs for similarity after standardizing them are compared. The degree of similarity between two ASTs can be measured using techniques like tree-edit distance. Following the distance calculations, a threshold would be set to determine whether a clone is considered. If a pair of code fragments has a distance below this threshold, then it may be considered a clone.

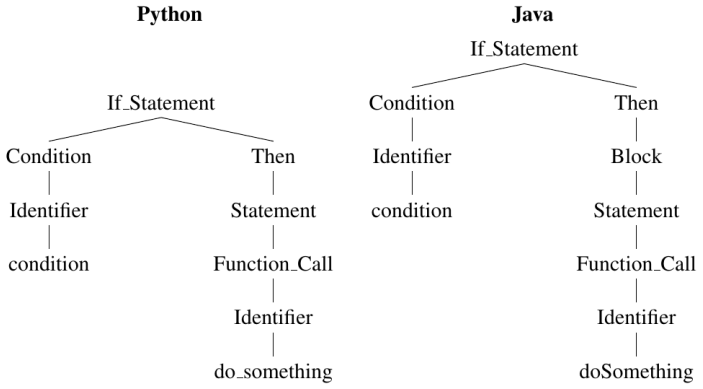
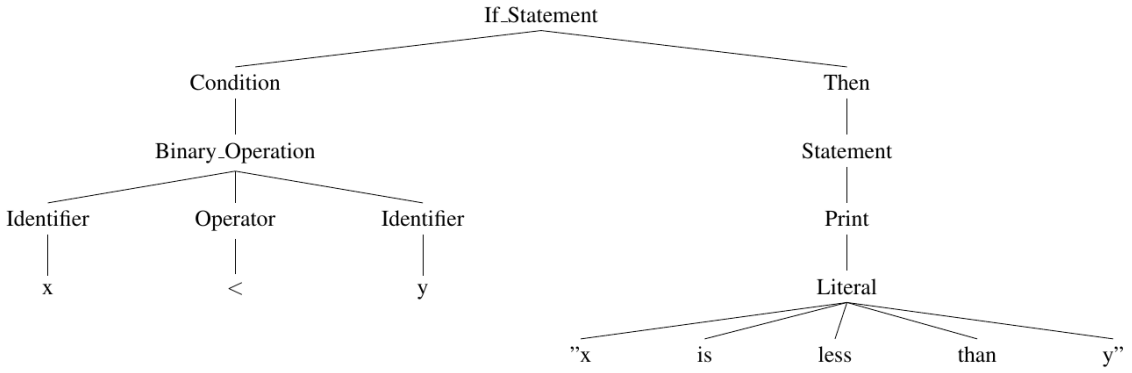


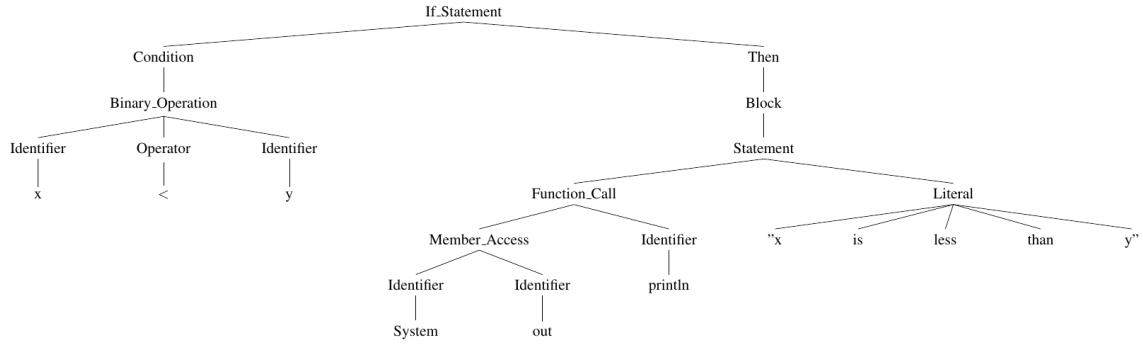
Figure 2.6: AST representation of the normalized Java and Python code.

An example how to compare the ASTs of both Java and Python agnostically is given. In the graph below Fig. 2.7, we can see that the tree structure is quite similar, but the names of the nodes differ due to the differences in grammar rules between Python and Java.

ASTs can be standardized by creating an abstraction that retains only the essential semantic information and discards the language-specific syntactical elements. As a result of



(a) AST of Python program.



(b) AST of Java program.

Figure 2.7: AST representation of a simple `if` program in Java and Python.

this abstracted AST, “if” statements can be represented in a way that is compatible with both Python and Java. The original ASTs can be compared directly to detect clones by converting them into this abstracted form. It is important to remember that this is a simplified example. ASTs are even more complex and detailed than actual programming languages. This should give a sense of how language-specific ASTs might be abstracted so that they can be used to detect cross-language clones. To illustrate how the abstraction in Fig. 2.8 was carried out, Python’s `Print` statement and Java’s `System.out.println` method call, for example, both print messages to the console. A generic `Print` node could represent both of these op-

erations in an abstracted AST node. Thus, the abstracted ASTs of Python and Java codes can be directly compared and cloned. After abstraction, each abstracted code fragment in Python and Java is compared, so two abstracted ASTs are compared. Finally, tree-distance algorithm is applied on the abstracted trees for both Python and Java.

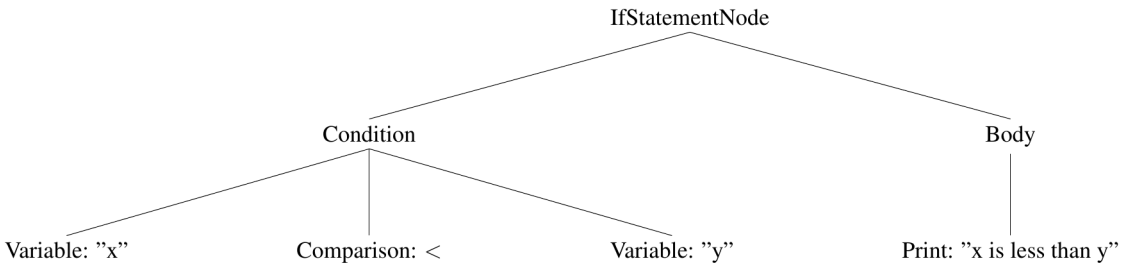


Figure 2.8: AST abstracted representation of the Java and Python code.

This abstraction process reduces source code from different languages (in this case, Python and Java) to a common intermediate representation. The result is a set of abstracted ASTs for each piece of source code, independent of the original languages' syntax and grammar. By comparing these abstracted ASTs directly, clones can be identified. Different techniques can be used to identify similar or identical structures in abstracted ASTs, such as tree matching algorithms. After abstraction, two abstracted ASTs for the Python and Java "if" statements are produced:

- Python AST abstracted.
- Java AST abstracted.

Due to the identical logic between the abstracted ASTs, a clone detection algorithm can recognize that these two pieces of code are in fact clones of each other, despite being written in different languages. As a result of this process, we are able to detect cross-language clones.

The development of an effective and accurate abstraction process requires a deep understanding of the syntax and semantics of the programming languages involved. To transform one abstracted tree into another, tree distance algorithms quantify the minimum number of operations (e.g., insertions, deletions, and substitutions). Zhang-Shasha and Ted Dijkstra are two common tree distance algorithms. The abstracted ASTs of two code fragments can be considered clones if the distance between them is below a certain threshold. Choosing the right threshold can affect the clone detection's precision and recall significantly. It is, however, possible that tree distance algorithms are computationally intensive, especially when dealing with large ASTs. It is possible to speed up the comparison process by using techniques such as tree hashing and tree fingerprinting. The process is quite complex with many intricacies, but that's the general concept. Last but not least, this approach assumes the abstraction process can preserve the semantic information of the code while eliminating the differences in syntax. In reality, creating such an ideal abstraction process is very challenging and is an active research topic [23, 38, 39]. DECKARD is a system for detecting code clones based on ASTs. The article discusses the challenges of using ASTs for code clone detection and the importance of abstraction [38]. CCFinder also abstracted away many language-specific syntax details prior to comparing two code fragments [39]. Svajlenko et al. started recently providing insights into the challenges of code clone detection across multiple projects and languages [72].

2.1.5 Challenges in Cross Language Clone Detection

In order to detect cross-language clones, there are a number of challenges that must be overcome. Due to the fact that different languages have different syntaxes and semantics, and different means of expressing similar functionality, language differences have a significant impact on these challenges. Python and Java can differ in syntax (form) and semantics (meaning) even for the same functionality. As a result, it is difficult to compare their Abstract Syntax Trees (ASTs). It is not straightforward to compare the AST of a Python code snippet

with the AST of a Java code snippet, such as `ASTPy`$\neq$ `ASTJava`. Here, $\neq$ signifies the difference in tree structures due to syntax differences. As discussed in Section 2.1.4, syntax can be abstracted in order to isolate semantics so that it's easier to compare code fragments.

The same functionality can be expressed in many different ways in different languages. It is difficult to define a universal abstraction function that can capture all equivalent code fragments due to these variations. Suppose, for example, a simple functionality that would check whether a number num was even is considered. In Python, it might written as $num\%2 == 0$, whereas in Java, it could be written as $(num\&1) == 0$. As a result of the differences in these constructs, different ASTs are created. Both of these expressions check for evenness, but they use different operations and thus produce different ASTs. The challenge here is defining a function A such that $A(\text{ASTPy}) = A(\text{ASTJava})$, where ASTPy is the AST of $num\%20$, and ASTJava is the AST of $(num\&1) == 0$.

First, for loop constructs, differences between Java and Python are normalized. Unlike Java, Python has a unique loop construct, the for-each loop. In Python, iterating through an array is usually done with `for i in arr:`, while in Java it's done with `for(int i=0; i<arr.length; i++)`.

Second, at exception handling level, unlike Python again, Java has checked exceptions that must be declared or handled. Because Python does not have try-catch blocks, the same piece of code would have one in Java. Because of this, their ASTs are different, and a tree standardization mechanism would be required to handle this. When a method throws a checked exception, the compiler enforces a rule: either the method must catch the exception and handle it (using a try/catch block), or it must declare the exception, signaling any methods calling it to handle this exception. The purpose of this safety feature is to ensure that every checked exception will be caught and handled. The rule is enforced at compile-time, so if it is not followed it, the program will not compile. Python, however, does not have this rule. Exceptions can also be caught and handled in Python. Despite this, Python does not differentiate between checked and unchecked exceptions. Python treats all exceptions equally.

Furthermore, Python does not require exceptions to be caught at compile time. Even if exceptions thrown by Python methods are not thrown, your program will still compile and run. As a result, Python is more flexible, but the programmer must remember to catch exceptions as well. This difference in how Python and Java handle exceptions makes clone detection difficult when converting Python and Java code into Abstract Syntax Trees (ASTs). Because Java has try/catch blocks, even if two methods do the same thing in both languages, the ASTs for these methods will look different because of the try/catch block. It may be harder to detect that these methods are clones because of this. In order to overcome this problem, the clone detection tool must be designed to account for these differences in language semantics. Below is checked and unchecked exceptions in Java Listing 2.5.

```
1  //Checked exceptions
2  void readFile() throws IOException {
3    // code that may produce an IOException
4  }
5
6  //Unchecked exceptions
7  void divideByZero() {
8    int x = 1 / 0; // This will throw ArithmeticException at
    runtime
9  }
```

Listing 2.5: Checked and Unchecked Exception in Java.

A checked exception is an exception that can be thrown by a method or constructor, but they are not `RuntimeExceptions` or Errors, and they must be declared in the throws clause of the method or constructor. In contrast, unchecked exceptions do not need to be declared in a method or constructor's throws clause. At compile-time, they are unchecked, so the compiler does not require you to handle them. Unchecked classes include Error and its subclasses, as well as RuntimeException and its subclasses. Also, NullPointerException, ArrayIndexOut-

OfBoundsException, and finally ArithmeticException. Unlike Java, Python does not have the concept of checked and unchecked exceptions. The Python interpreter does not require methods to catch or specify exceptions, which means all Python exceptions are technically unchecked. The following example illustrates this concept. Imagine we are writing a function to divide two numbers. A division by zero raises an exception in both Java and Python. A Python example of such a function can be found below in Listing 2.8. Due to the fact that we are not catching ZeroDivisionError in the function, it will bubble up to the calling code if b is zero. The program will terminate if the calling code fails to handle the exception as well. In the checked Python version, if b is zero, a ZeroDivisionError is raised. The code in the try block stops running, and control moves to the except block, which prints an error message and returns None.

```
1  //Checked exceptions
2  def divide(a, b):
3  try:
4    return a / b
5  except ZeroDivisionError:
6    print("Error: Dividing by Zero is not allowed")
7    return None
8
9  //Unchecked exceptions
10 def divide(a, b):
11   return a / b
```

Listing 2.6: Checked, which technically is not there in Python, and Unchecked Exception in Java.

Thirdly, lambdas and anonymous functions are edge cases an AST abstractor must deal with. Anonymous functions are treated differently in Python and Java. In Java 8, lambdas were introduced instead of first-class functions, which have been in Python for years. Fur-

thermore, they are used and declared differently. There can be complex edge cases when two ASTs have very different representations of similar functionality. The challenge is to define an abstraction function A that captures these differences, and a distance function d that measures similarities accurately. Examples of how to use lambdas (anonymous functions) in both Python and Java can be considered. Both languages use lambdas to define functions that can be used to transform a list of numbers - the same purpose. The way these lambdas are used, however, is quite different. To create a new list, Python uses the map function and a lambda, while Java uses the stream API. Despite the same resulting functionality (creating a list of squared numbers), the ASTs of the two languages represent this functionality differently. As a result of Python's syntax and Java's language features, this occurs. As a result, comparing ASTs directly or standardizing them for comparison can be challenging. In addition, Python lambdas can be composed of any number of expressions and re-used later, unlike Java lambdas, which can only be composed of one expression or a method reference. Different lambdas in Python and Java can result in different AST structures, even for code doing the same thing, and this presents another challenge for cross-language clone detection.

```python
# Define a list
numbers = [1, 2, 3, 4, 5]

# Use a lambda to square each number in the list
squares = list(map(lambda x: x ** 2, numbers))

# Print the list of squares
print(squares)  # Outputs: [1, 4, 9, 16, 25]
```

Listing 2.7: Lambda function in Python.

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
```

```
4
5 public class Main {
6   public static void main(String[] args) {
7     // Define a list
8     List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
9
10    // Use a lambda to square each number in the list
11    List<Integer> squares = numbers.stream()
12    .map(x -> x * x)
13    .collect(Collectors.toList());
14
15    // Print the list of squares
16    System.out.println(squares);  // Outputs: [1, 4, 9, 16, 25]
17  }
18 }
```

Listing 2.8: Lambda function in Python.

Finally, a bunch of other things make cross language clone detection difficult is their libraries and API usage, type system, runtime environment and resource managements. Python is dynamically typed, and Java is statically typed. While it's easier to handle differences in types by abstracting the AST and strip out types, others are not easy to handle. Python and Java handle multi-threading differently. For differences in runtime environment, an example of creating and starting a new thread in Listings 2.9 and 2.10 is considered. Although both code snippets print 0-9 in a new thread, their ASTs will be very different because Python and Java handle threads differently.

```
1 import threading
2
3 def print_numbers():
4 for i in range(10):
```

```
    print(i)

# Create a thread
t = threading.Thread(target=print_numbers)

# Start the thread
t.start()
```

Listing 2.9: Thread Function in Python.

```
public class Main {
  public static void main(String[] args) {
    // Create a thread
    Thread t = new Thread(new Runnable() {
      public void run() {
        for (int i = 0; i < 10; i++) {
          System.out.println(i);
        }
      }
    });

    // Start the thread
    t.start();
  }
}
```

Listing 2.10: Thread Function in Java.

Python and Java handle resources differently. In Python, the `with` statement is used to manage resources, which automatically closes them when they are no longer needed. Java, on the other hand, manually opens the resource, uses it, and then closes it. These differences in re-

source management would result in significantly different ASTs for these two pieces of code in Listings 2.11 and 2.12.

```
# Python uses the 'with' statement for resource management
with open('file.txt', 'w') as file:
file.write('Hello, World!')
```

Listing 2.11: Resource Management in Python.

```
import java.io.FileWriter;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    try {
      FileWriter writer = new FileWriter("file.txt");
      writer.write("Hello, World!");
      writer.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Listing 2.12: Resource Management in Java.

2.1.6 Difficulty in Standardizing Some Common AST Features in Cross-Language Setting

To answer the first research question as to why common features like threading in Python and Java ASTs cannot be detected, an example is used. The way you express threading in Python and Java is heavily influenced by the language's design, features, and standard libraries. "If statements" are universal across most programming languages. If a condition is

true, it performs an action. Since "if this, then do that" is a simple, direct operation, this basic semantic unit can be abstracted quite easily. ASTs typically consist of an if node, a condition and a body, respectively. However, the concept of a "thread" is not as simple or universal. An operating system and a language's runtime system define a thread at a lower level. Depending on the language's design, the way it interacts with threads can vary significantly. A thread in Python is started by calling a function in the threading module. The function takes another function as an argument and starts it in a new thread. That is very much in line with Python's design as a scripting language that allows functions to be passed around as arguments. Java threads are started by instantiating Thread objects. `Runnable` objects are passed as arguments to this object, and their `run` methods are executed in a separate thread. A design like this reflects Java's object-oriented nature, which ties actions to objects. Standardizing these two threading constructs in an AST would require understanding the semantic meaning of "start a new thread and run this code in it". The AST would then need to be able to represent this meaning in a language-independent manner. There are many aspects to threading, including concurrency control, thread lifecycles, inter-thread communication, and exception handling. Python and Java handle these aspects differently, so finding a common semantic representation is difficult. With Python, the lifecycle of a thread is handled by the garbage collector, which automatically reclaims thread resources when it finishes execution and is no longer referenced. Programmers in Java have greater control over the lifecycle of a thread, and can even forcefully stop or suspend a thread, although this is generally not recommended.

In addition, Python and Java handle exceptions differently. When an unhandled exception occurs in a thread in Python, the thread will terminate, but the main program will continue running. Java's UncaughtExceptionHandler is invoked when an unhandled exception occurs in a thread, which may or may not terminate the whole program based on how it is implemented. In other words, even for the same high-level action of "starting a thread", the underlying semantics of thread lifecycle and exception handling can differ greatly between Python and Java. Standardizing the ASTs would require a representation that captures all

these semantics accurately, which is not trivial. Due to the complexity of threading and the significant differences in how it is handled in different languages, standardizing the AST representation of threading across languages is a very challenging task. In addition to capturing the syntactic differences, it is important to accurately capture the semantic differences as well.

As well as the basic constructs of starting a thread, there are many other threading features that differ between Java and Python, including synchronization primitives (such as locks, semaphores), inter-thread communication (such as condition variables, queues), and so on. With Python, each thread can run on a separate core because of Python's Global Interpreter Lock (GIL). For similar multi-threading functionality, this could lead to different coding practices and ASTs. If GIL is influencing the way Python code is written, how would it be abstracted away? The GIL in Python is lost when the ASTs for Python code snippet that is to be compared with AST of Java code snippet for clone detection are abstracted. Both ASTs create multiple worker threads and wait until they are all completed after abstracting away the language-specific details. Despite this, the actual execution semantics under the hood are quite different due to Python's GIL. As a result, it is difficult to compare the two abstracted ASTs meaningfully.

Now another question is considered. By abstracting away the specific details of thread implementation and focusing solely on the presence of threading in Python and Java code, syntax code can be focused but not semantics of code snippets that contain threads? If code clones from the perspective of their functionality are strictly considered, then abstracting semantic details of thread is a valid approach. If whether a piece of code performs something functionally similar is only the interest, the high-level concept of a "thread" can indeed be abstracted and used as a comparison basis. In reality, however, thread semantics can differ significantly between languages, affecting the details of how that functionality is implemented. In Python, for example, multi-threading is quite different due to Global Interpreter Lock (GIL) while in Java, parallelism is real. In other words, even if two pieces of code, one in Python and one in Java, use "threads" in some way, their specific uses, management, and in-

teraction with the rest of the program can differ greatly. As a result, these differences will be reflected in the ASTs. When implementing a multi-threaded program that shares data across threads, Python and Java might handle synchronization differently. Unlike Java, Python does not require explicit locks to synchronize data access, because of the GIL. Due to this, the Java AST has nodes for lock acquisition and release, but the Python AST does not. In such cases, a simple abstraction of the concept of "thread" might not capture these nuances. It is possible that the abstracted ASTs look similar since they both have "thread" nodes, but the Python AST does not reflect the actual differences between the source codes due to the lack of synchronization nodes. In some cross-language clone detection situations, abstracting away the details of "thread" may be sufficient, but not in all cases. In clone detection, the choice of abstraction process is largely determined by the level of granularity needed. It is essential to use a more detailed abstraction process to capture the finer details of code that is very similar. However, this work is only interested in high-level functionality.

2.2 Neural Networks

2.2.1 LSTM Encoder Decoder

A neural network represents a series of methods that mimic the way the human brain functions to identify underlying relationships in a set of data. Using neural networks, for example Siamese network which is an architecture of energy-based models (EBM) [50], in our approach is primarily due to their high performance on similarity tasks, such as searching for similar images [22], and text [58]. It is possible for recurrent neural networks (RNN) to store representations of recent input events in the form of activations by using their feedback connections. An RNN is naturally suited for variable-length inputs, such as sentences, especially the Long Short-Term Memory (LSTM) model [34]. As part of the LSTM model, a hidden-state representation is sequentially updated, but a memory cell contains four components as well: the percentage to remember from long-term memory (forget gate), update the long-term memory (input gate), and finally is to update the short-term memory (output gate). In stan-

dard RNNs, backpropagated gradients are vanishingly small over long sequences due to the vanishing gradient problem over long sequences [59]. To solve this problem, the LSTM introduced a memory state c_t and three gates to control the flow of information over different time steps. Like the standard RNN, the LSTM sequentially updates a hidden-state representation.

2.2.2 Siamese Network

For the purpose of detecting sentence similarity, Siamese architecture with RNN was utilized [27] similarly to the Siamese architecture used in this work. In Siamese architectures, inputs (A, B) must generate the same output as inputs (B, A) (i.e. the similarity measure of A and B must be the same as that of B and A).

A Siamese architecture with LSTM can be used. The Manhattan LSTM model is outlined in Figure.2.9 [56], where h_a, h_b represents the hidden state output of LSTM of sentence 1 and LSTM of sentence 2 respectively and function that will take the distance between the 2 LSTM outputs. Each LSTM network processes one sentence in a pair. In this model, each input sentence is represented by word-vectors that are read in by the LSTM, and the final hidden state is used as a vector representation of each sentence. Consequently, semantic similarity is predicted based on similarity between these representations. This is very similar to this work in the Siamese model, but LSTM units, not RNN units are untilized.

In the Siamese network, the weights for the pair are shared so that symmetric outputs can be produced. If weights in the base network in Siamese architecture are not shared, getting the same output for either pair (A, B) or the pair (B, A) is not guaranteed. In the context provided, the goal is to ensure that the model produces symmetric outputs, i.e., if the model predicts a certain output for input (A, B), it should ideally produce the same output for input (B, A). However, when using a network with separate weights, this symmetry is not guaranteed. To improve symmetry, the suggestion is to train the model using both $(A, B))$ and (B, A) pairs of samples with equal frequency. Despite that training time is doubled (and therefore training steps are doubled), the network output is still not guaranteed to be sym-

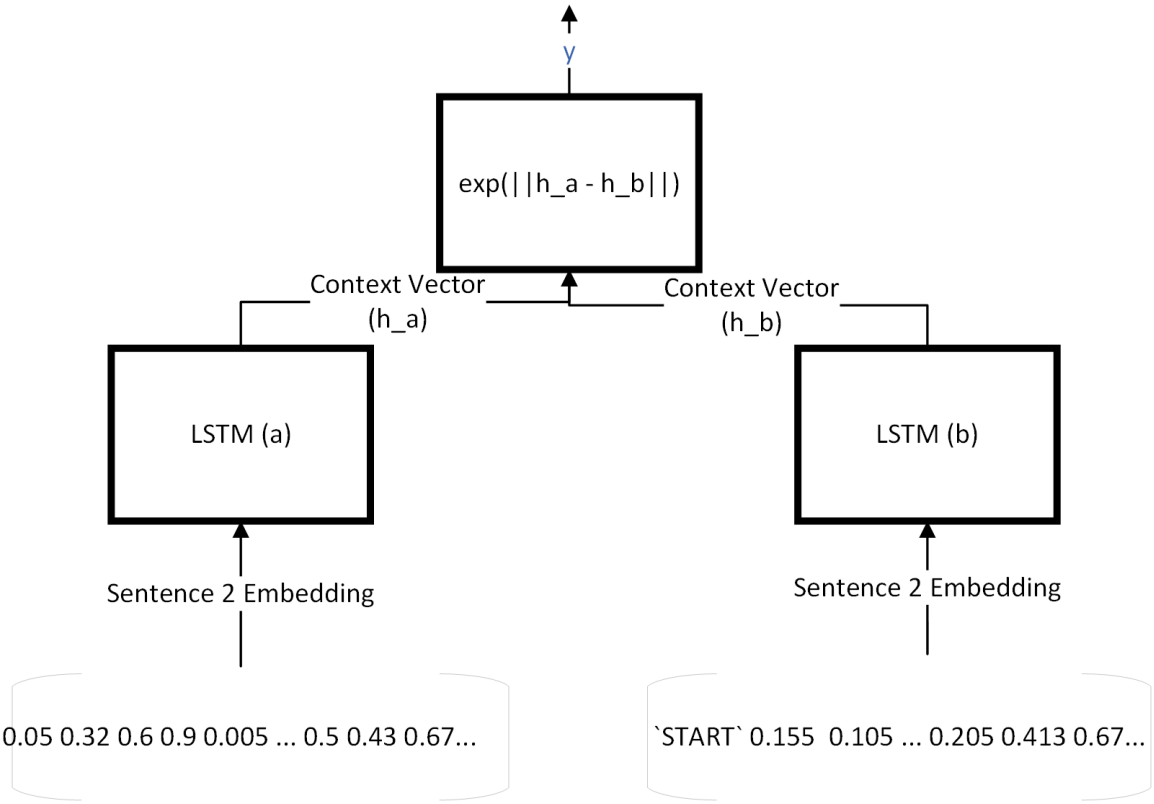


Figure 2.9: Siamese archeticture used with LSTM to measure similarity.

metric. To summarize, Sharing weights with the base network ensures that the network's response ((A, B) has the same output as (B, A)).

The Siamese neural network consists of multiple identical networks with the same weights and trained simultaneously. This work used only 2 parallel network trained on the same weights. The primary goal of Siamese networks is to learn a similarity or distance metric between input pairs. This work uses a Siamese network with two identical subnetworks. Considering input pairs A and B, let w represent the shared weights of $f(A)$ and $f(B)$.

Each subnetwork produces a feature vector $f_w(A)$ and $f_w(B)$ corresponding to an input pair (A, B). Similarity or distance between these feature vectors is typically the output of the Siamese network. Among the most common distances are the L2 distance (Euclidean distance):

$$d\left(A, B\right) = ||f_w\left(A\right) - f_w\left(B\right)||_2 \tag{2.5}$$

In the case of $A = B$, $f_w(A) = f_w(B)$ as both subnetworks share the same weights w. The symmetry property of the Siamese network output. For input pair (A, B), is:

$$d\left(A, B\right) = ||f_w\left(A\right) - f_w\left(B\right)||_2 \tag{2.6}$$

To prove the output is symmetric, $d(A, B) = d(B, A)$ needs to be satisfied.

$$d\left(B, A\right) = ||f_w\left(B\right) - f_w\left(A\right)||_2 \tag{2.7}$$

By the properties of the $L2$ norm, $||x - y||_2 = ||y - x||_2$ for any vectors x and y. So, $||f_w(A) - f_w(B)||_2 = ||f_w(B) - f_w(A)||_2$. Thus, $d(A, B) = d(B, A)$, which means flipping the input pair to either Python, Java or Java, Python is the same.

2.2.3 Attention Neural Networks

Attention neural networks came to resolve the issue of long range dependencies between sequence tokens in RNN network. Also, due ot recurrence nature of RNN networks where each timestep depends on the previous timestep, this prevents parallel computation [74]. Attention blocks can find, which input vectors has more weight in predicting output. Attention helps also with vanishing gradient problem as the attention network will do the computation for the input embedding simultaneously instead of doing it sequentially as in RNN, which was the source of vanishing or exploding gradients.

Equation 2.8 below represents attention mechanism att, where we want to find weighted sum of all values [74]. The weight is calculated as the similarity sim between a query Q and a key k_i. The whole equation can be interpreted as the expected mean of values.

$$att\left(Q, K, V\right) = \sum_i sim\left(Q, k_i\right) \times v_i \tag{2.8}$$

where the sim function gives a scaler $c \in \mathbb{R}$. It can be defined as the addition or the product between the query Q and one of the keys $k_i \in \mathbb{R}^d$. sim can take on the following forms:

- Dot product between Q and k_i as $Q^T k_i$ as in Loung attention [52].
- Scaled dot product as $\frac{Q^T k_i}{\sqrt{d}}$, where d is the dimensionlity of input key or query.
- General dot product $Q^T W k_i$, where W is a weight matrix learned through training.
- Additive similarity $w_Q^T Q + w_k^T k_i$ as in Bahdanau attention [8].

In general, the general dot product $Q^T W k_i$ is mostly used. During backpropagation, the neural network keeps updating weight matrix W that will span the solution space that will eventually give us the optimized space that maps queries to map queries to keys.

There are 3 different types of attention networks used in this work: 1) additive attention, 2) dot product attention, and 3) self-attention mechanism. The self-attention mechanism with a Siamese variant architecture is adopted. The first two attentions are explained in Section 3.4.

Self-attention accepts a sequence of input vectors and output vectors. Basically, any output vector is simply a weighted sum of the input vectors. For every output, a set of N weights and N input vectors is associated. The sequence of outputs will be created when we do this for each output. The weights are not model parameters shown in the above general attention mechanism, but it's computed as follows:

$$y_i = \sum_j w_{ij} x_j, \tag{2.9}$$

$$w_{ij}^{'} = x_i^T x_j, \tag{2.10}$$

$$w_{ij} = \frac{e^{w_{ij}^{'}}}{\sum_j e^{w_{ij}^{'}}} \tag{2.11}$$

where T is the transpose operation, and w_{ij} is defined as the $SoftMax$ function.

Without self-attention layer, each token in the embedding can only contribute to the output score independently from every other token. This is known as a bag of words model. The presence of a certain token in the embedding could indicate that the pair is not a clone, but with self-attention, it will look for surrounding tokens and capture another token that can could confirm that the pair of Java and Python embeddings are indeed clones. Without self-attention, the token 0.67 in Python embedding could be enough to tell that this embedding is for subtraction, but when with self-attention that looks around that token, we see that 0.67 is highly correlated with 0.005 in predicting the output.

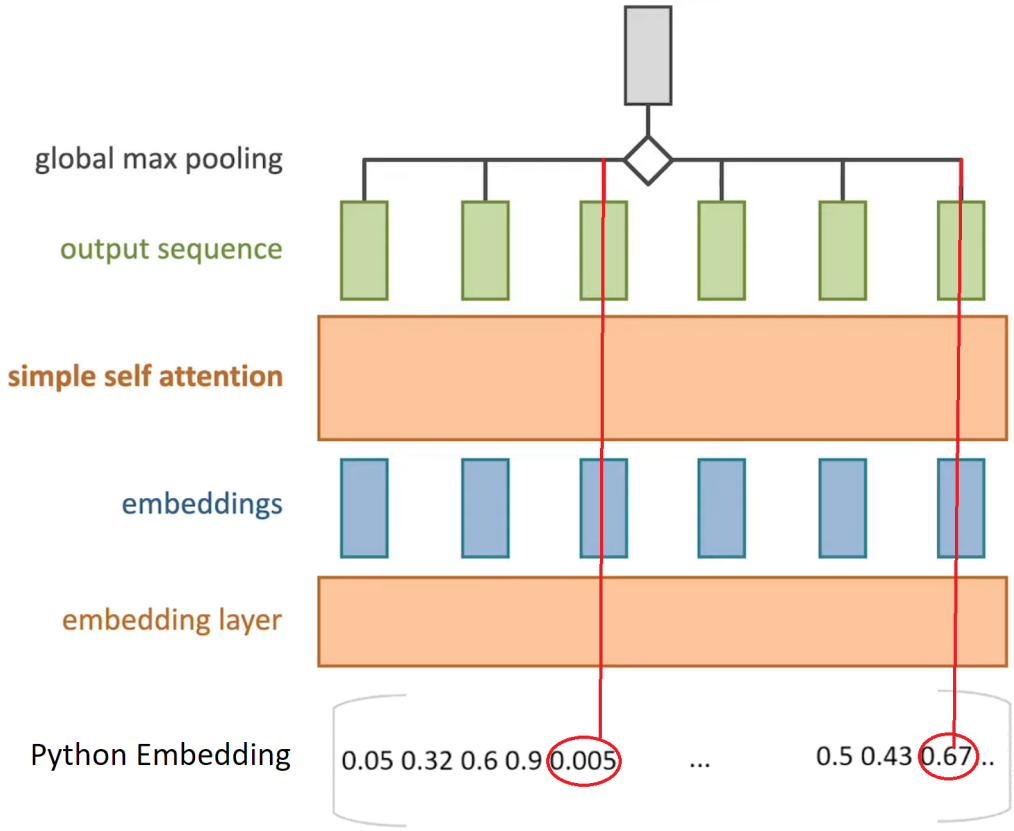


Figure 2.10: Classification model that takes an input Python InferCode embedding and then finds with self-attention which pars of input are correlated with each other.

2.3 Nearest Neighbor Search

In order to detect clones on a large scale, it is imperative to know how to search for nearby embedding representing a source code method in a system in order to accomplish this. A clone potential can be found by finding the nearest neighbors. Nearest neighbor search has been used in many domains like machine learning, data mining, image classification and in other scientific applications such as genome studies [25].

If the data dimensions is low, then KD-tree data structure is sufficient to query and retrieve potential clones. However, when the dimensionality of data is large as in our case, where each source code method is represented by a 100 dimensional vector, it becomes harder to find the clones.

2.3.1 K-Nearest-Neighbor (KNN) with Index for Large-Scale Clone Detection Search

Based on a similarity function, K-NN searches for vectors that are similar to a query vector. To speed up querying, an index is used along with the algorithm. Without indexes, a query has to scan every row in the table from the beginning to the end to find source code clone candidates.

Monitoring the validation and test dataset results will help select the parameter k representing the number of nearest neighbors carefully. A high k could lead to a high bias, while a low k could lead to a high variance. k should be set where the validation error is minimal based on the validation graph during training. k should be set where the validation error is minimal based on the validation graph during training. In high-dimensional data, what is the best distance function? Even though it only calculates the distance between the embedded InferCode and the dataset, it affects the accuracy of the training. With high-dimensional data, it's better to choose a small value p in Minkowski difference:

$$d\left(embed_1, embed_2\right) = \sqrt{\sum_{i=0} \left(embed_{1i} - embed_{2i}\right)^p} \tag{2.12}$$

Since the data used in this work is of high dimension nature (100 dimensional for each embedding), K-nn performs poorly [70]. The reason for that is due to the cost of distance function computation and difficulty to interpret distance closeness in high dimensions as the sum between vectors in high dimensional space increase and the space becomes more and more sparse. This alleviate this problem can be alleviated by using dimensionality reduction [7].

2.3.2 Approximate K-NN

To alleviate the curse of dimensionality of K-NN, approximate solutions can be adopted. Approximation algorithms can be adopted if searching for the exact solution becomes too expensive. Locality-sensitive hashing (LSH) is used in hashing to group similar items into high-probability buckets [71]. As the name implies, similar items should end up in the same buckets at the end. When it comes to querying and generating data, hash tables are efficient and easy to use. To find approximate nearest neighbors, the LSH method is used in this way.

Observed Mean Distance divided by Expected Mean Distance is the Nearest Neighbor Index. Based on a hypothetical random distribution, the expected distance between neighbors is the average distance between them. Clustering occurs if the index is less than 1. Competition occurs if the index is greater than 1.

2.4 Evaluation Metrics to Evaluate CLCD-I

The problem being addressed in this work is a binary classification problem as each pair is classified as either a clone or a disclone. In this work, deep learning models are evaluated using the Matthews Correlation Coefficient (MCC), F1 score and accuracy. Despite their common use, they have different mathematical formulations and properties. MCC is a performance metric that evaluates the quality of binary classification by taking into account true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). The range

of MCC is -1 to 1, with 1 representing a perfect prediction, 0 representing a random guess, and -1 representing a totally wrong prediction. MCC is defined as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)\,(TP + FN)\,(TN + FP\,(TN + FN)}} \tag{2.13}$$

An F1 score measures the rate at which precision versus recall is traded off in binary classification problems. The precision of a model is the proportion of true positive predictions among all positive predictions. Essentially, it tells us how many of the positive instances are actually true. Precision is calculated as:

$$Precision = \frac{TP}{(TP + FP)} \tag{2.14}$$

where TP is the number of true positives and FP is the number of false positives. True positive rate, also known as recall, is the proportion of true positive predictions among all actual positives. It tells how many actual positive instances were correctly identified by the model. Recall is calculated as:

$$Recall = \frac{TP}{(TP + FN)} \tag{2.15}$$

where TP is the number of true positives and FN is the number of false negatives. F1 is the harmonic mean of precision and recall. A harmonic mean is usually used when dealing with rates or ratios, as it gives more weight to lower values, resulting in greater penalties when precision and recall are extremely imbalanced. The formula for F1 score is:

$$F1 = 2 \times \frac{(\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})} \tag{2.16}$$

where Precision = TP / (TP + FP) and Recall = TP / (TP + FN).Using the harmonic mean, the F1 score allows you to compare the performance of different classifiers by capturing the trade-off between precision and recall. With a score of 1, perfect precision and recall are exhibited, while a score of 0 indicates the worst performance.

To illustrate how powerful F1 is for classifiers. The example of harmonic mean between 2 values is compared with the arithmetic mean. As a result, the harmonic mean of a set of values is just the reciprocal of the arithmetic mean of the reciprocals of those values. The harmonic mean of two values, a and b, is:

$$\text{Harmonic Mean} = 2 \times \frac{1}{\frac{1}{a} + \frac{1}{b}} \tag{2.17}$$

To see how the harmonic mean penalizes imbalances between precision and recall, how it gives more weight to lower values is examined. Imagine a classifier with a precision of 0.9 and a recall of 0.1. This is an extremely imbalanced value. As a result, the arithmetic mean would be:

$$\text{Arithmetic Mean} = 2 \times \frac{0.9 + 0.1}{2} = 0.5 \tag{2.18}$$

However, the harmonic mean (F1 score) would be:

$$\text{F1} = 2 \times \frac{2}{\frac{1}{0.9} + \frac{1}{0.1}} = 0.18 \tag{2.19}$$

The harmonic mean is significantly lower than the arithmetic mean, which is a reflection of the imbalance between precision and recall, which is reflected in the harmonic mean.

Classification algorithms, encompassing deep learning models, are frequently assessed using both accuracy and F1 score as evaluation metrics. These two metrics possess distinct mathematical formulations and characteristics, even though they are commonly employed for analogous objectives. Broadly speaking, accuracy is computed as the proportion of accurate predictions, which includes true positives and true negatives, relative to the entire set of predictions. The formula for calculating accuracy is as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.20}$$

where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives. Suppose the following values: $TP = 80, TN = 900, FP = 20, FN = 100$. The accuracy and F1 score are calculated.

$$\text{Accuracy} = \frac{80 + 900}{80 + 900 + 20 + 100} \approx 0.891 \tag{2.21}$$

$$\text{F1} = 2 \times \frac{0.80.444}{0.8 + 0.444} \approx 0.571 \tag{2.22}$$

In this example, the accuracy is 0.891, while the F1 score is 0.571. The lower F1 score indicates that there is an imbalance between precision and recall, which the accuracy metric does not capture.

The choice to select between different evaluation metrics for a deep learning model depends on several factors such: 1) Sensitivity to imbalanced datasets, 2) Emphasis on trade-off between precision and recall, and 3) Penalizing extreme imbalances. For example, in a spam email detection system, both false positives (non-spam emails marked as spam) and false negatives (spam emails not detected) should be minimized. The F1 score can provide a better overall evaluation of the classifier's performance in such situations, as it inherently balances the trade-off between precision and recall. In this specific study, given that our dataset is perfectly balanced and consists of 40,000 pairs of code embeddings learned by InferCode. Consequently, the selection between the F1 score and accuracy metrics is rendered less imperative. Both accuracy and F1 score can be used to evaluate the performance of CLCD-I model.

Before discussing AST aspect of Type III clones, why 0.7 threshold we discuss earlier is usually used in similarity score $S(A, B)$ needs to be understood. The balance between precision and recall in code clone detection will be examined. In Information Retrieval and Machine Learning, precision and recall are typically used to measure the performance of binary classifiers. Precision (P) is defined as the proportion of true positives (TP) to the sum of true positives and false positives (FP). False positives are code fragments incorrectly identified

as clones in clone detection, while true positives are code fragments correctly identified as clones. Recall and precision will be defined later. In clone detection, a false negative is a pair of code fragments that are clones but were not identified as such. There is usually a trade-off between precision and recall when trying to maximize both. In effort to increase recall (by identifying more true positives), the number of false positives is often increased, thereby reducing precision. On the other hand, if precision (by minimizing false positives) is aimed, some true positives are missing, thereby decreasing recall.

The trade-off between recall and precision more clearly can be illustrated using by increasing/decreasing the terms in the precision and reall equations. Recall can be increased by increasing true positives (TP) or decreasing false negatives (FN). The easiest way to accomplish this is to classify more code pairs as clones. In doing so, false positives (FP) are likely increased as and more pairs are incorrectly identified as clones. As FP increases, the precision equation's denominator increases, resulting in a decrease in precision. Conversely, if precision is increased by reducing false positives, fewer code pairs are classified as clones. The conservative approach in this work may indeed lead to a decrease in false positives (FP), but it may also lead to an increase in false negatives (FN) since some true clones. By increasing FN, the recall equation's denominator increases, decreasing recall value. A less conservative model, meaning more code pairs are classified as clones, should result in more True Positives (TP) by capturing False Negatives (FN). However, some pairs might be classified as clones that are not actually clones as a result of this, which would be False Positives (FP).

Consider the following example to illustrate the precision-recall trade-off with a codebase containing 1000 pairs of code fragments. Here are the current stats: 1) True Positives (TP): 200 pairs are correctly identified as clones, 2) False Positives (FP): 50 pairs are incorrectly identified as clones. 3)

True Negatives (TN): 700 pairs are correctly identified as not clones. 4) False Negatives (FN): 50 pairs are clones but not identified. Given these stats, your Precision (P) = TP / (TP + FP) = 200 / (200 + 50) = 0.8 and Recall (R) = TP / (TP + FN) = 200 / (200 + 50) = 0.8.

Now, suppose that recall is increased by classifying more code pairs as clones. The clone detection tool is adjusted to be more liberal in identifying clones. As a result, 20 of the previously False Negative pairs are now correctly identified as clones (True Positives). However, being less conservative also led to an additional 30 pairs being incorrectly classified as clones (False Positives). So, the updated stats are: 1) True Positives (TP): 220, 2)

False Positives (FP): 80, 3) True Negatives (TN): 670, 4) False Negatives (FN): 30. Now, your Precision (P) = TP / (TP + FP) = 220 / (220 + 80) = 0.733 and Recall (R) = TP / (TP + FN) = 220 / (220 + 30) = 0.88. In this example, it can be seen how increasing recall led to a decrease in precision. Even though it is managed to correctly identify more clones (increasing TP and decreasing FN, thereby increasing recall), this was at the expense of incorrectly identifying more non-clones as clones (assuming FP, thereby decreasing precision).

CHAPTER 3

RELATED WORK

3.1 Cross Language Clone Detection

There are two distinct approaches to detecting cross-language code clones. In Perez et al.'s model, token-level vectors are generated, clone detection models are trained, and actual code clone detection is performed. To generate token-level vectors, it uses a skip-gram algorithm based on Abstract Syntax Trees (ASTs) of Python and Java codes. For clone comparison, these vectors are fed into a Siamese supervised neural model. The model uses AST structures to define token context and aims to automatically learn the mapping between Python and Java code ASTs. In contrast, Nafi et al.'s CLCDSA model leverages both semantic and syntactic features of Java, Python, and C# code fragments. In order to capture similar metrics across languages, it selects nine frequently occurring features from a larger set. However, this model requires extensive pre-processing, such as API documentation inclusion and skip-gram based API call similarity filtering before clone detection begins, making it less of an end-to-end solution compared to Perez et al.'s approach.

3.1.1 Tree-based skip gram model

Using a Siamese neural network architecture, Perez et al. propose a novel method for cross-language clone detection [60]. To encapsulate structural information from the source code, the system transforms input code snippets into Abstract Syntax Trees (ASTs). In order to capture the semantic information that is inherent to each token, these ASTs are fed into a tree-based skip-gram model that generates dense vector representations for each token. After this stage, an LSTM layer is applied to the sequence of token embeddings. In this LSTM layer, the sequence is analyzed and tokens are ordered according to their dependency and the cumulative information is encapsulated into a final state vector that represents the cumula-

tive information. The output of this LSTM is then passed through a hash layer, in which the vector is converted into a fixed-size hash code, allowing direct comparison between codes originating from different programming languages. In order to determine whether the corresponding code snippets are likely to be clones of each other, this system is concluded with a classifier that examines these hash codes in order to determine the likelihood that they are identical.

Basically, in Perez et al. work, they propose a semi-supervised machine learning approach that works with programming languages with similar syntax, since conventional comparison-based methods cannot be applied directly in this situation. Typically, comparison-based methods detect clones by comparing the syntax or structure of code fragments and looking for similarities or exact matches. When it comes to cross-language detection, these approaches aren't as effective as they are within a single language with consistent syntax and structure. This is due to the fact that different programming languages have different syntax and structures, even if they perform the same function. Depending on the programming language, variables, flow control, and data structure can differ significantly. Because of this difference, comparison-based methods cannot identify clones across languages because the raw code or syntax they are comparing won't be the same, even if the logic is identical. Consider two code fragments, f_1 in language L_1 and f_2 in language L_2. These fragments can be represented as Abstract Syntax Trees (ASTs), A_{f_1} and A_{f_2}, respectively. An AST comparison-based method would, in a single-language example, seek a direct mapping, say M, between f_1 and f_2 (e.g., $M : A_{f_1} \rightarrow A_{f_2}$). It may be possible to argue that f_1 and f_2 are identical if such a mapping exists. The same direct mapping might not exist in a cross-language scenario, however, since the languages L_1 and L_2 have different syntax and structures. In other words, even if f_1 and f_2 are logically identical (i.e., they are clones), a direct mapping M between A_{f_1} and A_{f_2} might not be found due to the differences in L_1 and L_2. This is why comparison-based methods struggle with cross-language clone detection. The authors' proposed method bypasses this issue by learning a transformation $T : L_1 \rightarrow L_2$ (or

$T : A_{f_1} \rightarrow A_{f_2}$) that can effectively translate the syntax and structure of L_1 to L_2 (or A_{f_1} to A_{f_2}), making it possible to compare code fragments in different languages.

The skip-gram model, originally proposed by Mikolov et al. [49] for natural language processing, is designed to predict the context of a word given the word itself. It is fairly common for skip-grams to be used for natural language processing to define a word's context as the words around it in a sentence, within a certain window size, in a typical application of skip-grams. Nevertheless, in the case of a tree-based skip-gram model, the context of a token (which corresponds to a node in the Abstract Syntax Tree (AST)) is determined by its location in the AST rather than its location in a linear sequence of tokens. They take the advantage of AST structures to define the context of a token in AST instead of windows used in natural language processing. A node context is customizable in the skip-gram model to include as many children as possible. The context also includes one parent, which establishes a child-parent relationship in the context. The context of a node is defined by the ancestors, descendants, and siblings of the node within a given window which is amenable. Suppose that Python and Java code examples are denoted as f_{py} and f_{ja} respectively. ASTs can be represented as A_f for each function f. In their system, they aim to find a mapping $M : A_{f_{py}} \rightarrow A_{f_{ja}}$ that describes how the ASTs of f_{py} and f_{ja} correspond to each other. Such a mapping M can be learned automatically by the proposed system, which has a significant advantage over manual rule creation. Consider an AST where each node n has a token and its context is defined by the nodes related to it in the tree. For example, a node's context might consist of its parent node, its sibling nodes, or its child nodes. A function context(n) that, given a node n, returns the set of context nodes for n is defined. Also a function token(n) that, given a node n, returns the token of the node is defined.

Given a program represented by an AST with nodes N, the objective of the tree-based skip-gram model can be formalized as follows:

$$\text{Objective} = \sum_{n \in N} \sum_{m \in \text{context}(n)} \log P(\text{token}(m) | \text{token}(n)) \tag{3.1}$$

where $P(\text{token}(m)|\text{token}(n))$ is the probability that token m appears in the context of token n, where both n and m are in the set of AST tokents either for Java or for Python. This probability is modeled using the softmax function:

$$P(\text{token}(m)|\text{token}(n)) = \frac{\exp(\mathbf{v}'^{T}_{m}\mathbf{v}n)}{\sum_{j\in V}\exp(\mathbf{v}'^{T}_{j}\mathbf{v}n)} \tag{3.2}$$

where $\mathbf{v}_n$ is the vector representation (or "embedding") of the token of node n, $\mathbf{v}'_m$ is the output vector representation of the token of node m, and V is the set of all possible tokens. The model parameters, which are the vector representations of the tokens, are then learned by maximizing the objective function using techniques such as stochastic gradient descent. By learning vector representations of tokens, this model detects clones in code as they capture semantic similarities between code fragments.

To explain how LSTM encoder layer in Perez et al. works as in 3.1, assume that the tree-based skip-gram model give output $[0.1, 0.3]$ for function name function name (or `addNumbers`) is represented by $[0.1, 0.3]$, `x` is represented by $[0.2, 0.4]$, and so on. Let us see more carefully how LSTM layer performs calculations for an input function. This layer processes the sequence of token embeddings, capturing the order and dependencies among the tokens. Each LSTM cell takes in a token embedding and its own previous state, performs some calculations, and outputs a new state. This new state is then passed to the next LSTM cell, along with the next token embedding. In this way, each LSTM cell captures the information of the current token and all previous tokens. For three LSTM cells, and their output states are as follows:

- LSTM cell 1 (processing the function name `addNumbers`): $[0.5, 0.6]$.
- LSTM cell 2 (processing `x`):$[0.7, 0.8]$.
- LSTM cell 3 (processing `y`):$[0.9, 1.0]$.

The final output of the LSTM layer would be the state of the last LSTM cell, $[0.9, 1.0]$, which captures the information of all tokens.

```
1  public class Main {
2    public static int addNumbers(int x, int y) {
3      return x + y;
4    }
5
6    public static void main(String[] args) {
7      int result = addNumbers(5, 10);
8    }
9  }
```

Listing 3.1: Checked and Unchecked Exception in Java.

Perez et al.'s system uses the hash layer to translate high-dimensional LSTM output vectors into fixed-size hash codes. LSTMs produce a state vector for each code snippet based on all the syntactic and semantic information inferred from the original code snippet through previous layers, which is a dense and high-dimensional representation which encapsulates all the syntactic and semantic information derived from the original code. In order to make these high-dimensional vectors more manageable, the hash layer transforms them into a fixed-size format in the form of a low-dimensional vector. In order to keep the vital information that distinguishes different code snippets from each other during this process, it is important to perform this transformation in a manner that maintains the essential information. As a result of Locality-Sensitive Hashing (LSH), a method that ensures that similar inputs generate similar hashes, they can achieve this goal. In LSH, input vectors are mapped into buckets using a set of hash functions. When vectors are similar (i.e., close-by in the high-dimensional space) then there is a high probability that they are mapped to the same bucket due to the similarity between them. In the context of Perez et al.'s system, the LSH ensures that the LSTM output vectors of similar snippets of code, possibly indicating code clones, are translated into

hash codes that are identical or closely similar to the code snippets. Last but not least, the fixed-size hash codes, which are significantly lower in dimensionality than the LSTM output vectors, allow efficient comparison of different snippets of code that may differ in size. Those snippets form the input for the final classifier layer which determines whether or not the given snippets can be classified as clones. Specifically, the LSH-based Hash Layer used by Perez et al. cannot be trained. The layer applies a function (the LSH function) to the output of the preceding LSTM layer. During the training of the model, the parameters of this function are typically not updated. The most common types of LSH functions are based on random projections. Here is a simple version of such a function:

- Pick a random vector w in the N-dimensional space. Each component of w is picked from a standard Gaussian distribution, meaning it's a random number with a mean of 0 and a standard deviation of 1. This vector w serves as a "projection line" in our high-dimensional space.
- Pick a random offset b from the range $[0, r]$, where r is the magnitude of w. This offset b will serve to "shift" our projection line randomly.
- For any input vector v (from our LSTM output), compute the dot product $w \cdot v + b$, which essentially projects v onto the line defined by w.
- Take the sign of the result. This is our hash code. In other words, the hash function $h(v)$ is defined as sign$(w \cdot v + b)$.

The last layer of Perez et al. does the classification. This layer takes in the hash codes of two code snippets and outputs a probability that the snippets are clones. Let the hash code of the Python snippet be $[1, 1]$ and the Java snippet be $[1, 1]$. The classifier might compare these codes directly and output a probability of 1 (or 100%), indicating that it's certain these two snippets are clones. To train this classification layer, Perez et al. used a set of labeled data, i.e., pairs of code snippets that are known to be clones or not. The classifier would be trained to minimize the difference between its output and the true labels of the input pairs.

This could be done using a loss function like binary cross-entropy loss, which is commonly used for binary classification problems.

3.1.2 Feature-Based Cross-Language Clone Detection

Nafi et al. adopted a CLCDSA model to detect cross-language code clones in Java, Python, and C# utilizing semantic and syntactic features of code fragments. Based on the results of a heuristic and manual study, they were able to extract 9 out of 24 total features in Saini et al.'s [64], and based on the heuristic study, they found that these 9 features are more common in cross-language scenarios. Furthermore, these nine features yield similar metrics when compared between different but functionally similar programming languages. Although the model that Perez et al. developed achieved higher scores than the one used by Perez et al., there are a number of preconditions to meet. It was found that, in addition to exhaustive feature selection, API documentation was also included as part of the CL model as part of the search for clones, which greatly increased the workload for the CL model. Moreover, they demand that pairs of data have to undergo a similarity filtering process based on the skip-gram model of similarity before the clone detection process begins in order to assess the possibility of being identical to each other. In conclusion, this is not an end-to-end approach and extensive feature processing must take place before the data can be used in the analysis.

Nafi et al. implemented a feature-based model instead of an AST-based model used by Perez et al. They did not directly employ features from the Abstract Syntax Tree (AST) of the source code, instead, their model extracted certain syntactic and semantic features from the source code, including some that could be derived from an AST, like the number of methods, loops, and conditions, but the features themselves are not AST nodes or structures. As a result of this deviation, the CLCDSA model can detect cross-language clones based on both syntax and semantics, rather than only structural similarities between languages.

In total, the entire methodology can be categorized into three principal steps, namely: Feature Extraction, Filtering Process, and Clone Detection. As part of the Feature Extrac-

tion process, nine pivotal characteristics of the source code are extracted both syntactically and semantically from the source code. A number of factors are taken into consideration in this process, including the length of the code, the number of methods, loops, conditions, API calls, variables, the cyclomatic complexity of the code, normalized compression distance, and the number of API calls. In the next stage of the process, these features will feed into a Deep Neural Network (DeepNN) for further processing. There is an action filter in place as part of the Filtering Process that makes use of API call documentation across different programming languages and, as a result, eliminates pairs of source code that do not contain common API calls. There is a substantial reduction in the number of comparisons that are needed in the ensuing phase of Clone Detection as a result of this. During Clone Detection, the source code pairs that survived the filter are processed through a DeepNN network. For each pair of features extracted earlier, a vector representation is learned by the DeepNN network. If the cosine similarity between these vectors exceeds a predefined threshold, the source code pair is considered cloned.

There are nine key features of CLCDSA, both in terms of syntax and semantics, which are taken from both the source code and the documentation. With the help of an action filter that is based on API call documentation, the model is able to filter out non-clone code pairs from the model. Based on the nine features, the DeepNN network is then used to learn a vector representation of the code pairs using the nine features that make up each code pair. CLCDSA is unique in its ability to detect clones on-the-fly without having to undergo any pre-training procedure in advance. As compared to other models, CLCDSA exhibits superior performance in terms of precision, recall, and F-measure when compared to other models. It is evident that its high efficacy and potential for handling cross-language clone detection challenges demonstrates its high efficiency and potential.

DeepNN consist of an input layer, several hidden layers, and an output layer. In each layer, there are numerous interconnected nodes called neurons, and the connections between these neurons determine the network's overall output. For every piece of source code, Nafi

et al. create a 9-dimensional input vector, where each dimension corresponds to one of the selected features. These features may include the frequency of API calls, loop constructs, conditional constructs, exception handling constructs, IO operations, the use of third-party libraries, multi-threading usage, data structure usage, and recursion usage. Feature vectors are then fed into DeepNN after feature extraction. In the first hidden layer, neurons perform a linear transformation (characterized by a weight and bias) on these inputs, then apply a non-linear activation function (such as a ReLU or sigmoid function) to the result. This process is then repeated across all layers of the network. Typically, the output layer provides the desired format for the network's predictions - in this case, the similarity score between two pieces of source code. In order to minimize the difference between the network's output and the known similarity scores for these pairs, pairs of code snippets are fed into the network and weights and biases are adjusted using a process called backpropagation. After the DeepNN has been trained, it can be used to compare new code snippets: each snippet's features are extracted and analyzed by the network to create a similarity score, which can then be used to determine if the code snippets are similar.

To sum up, unlike other skip-gram models, Perez et al.'s tree-based model can distinguish cross-language code clones based on structural similarity of abstract syntax trees. Thus, it can be generalized over a wide range of programming languages. Since it operates at the level of ASTs, it is less sensitive to cosmetic differences between code snippets, such as variable naming or formatting, which are irrelevant for determining code functionality. This approach, however, has a relatively low precision rate, which is one of its main disadvantages. Even if two code snippets have similar ASTs but perform different functions, the model can still classify them as clones. Moreover, it requires a threshold to determine whether two ASTs are similar enough to be considered clones, which may not generalize well across datasets. Nafi et al.'s CLCDSA approach, however, considers both syntactic and semantic aspects of a code. It can determine if two code snippets perform the same operations even if they are written in different languages and have different structures by using feature vectors, including

API call vectors. In this way, it is a robust tool for detecting functional clones. Also, using an Action Filter reduces the number of comparisons required, boosting its scalability. Furthermore, the approach exhibits superior performance compared to existing models in terms of precision, recall, and F-Measure, indicating that it effectively balances the trade-off between clone detection sensitivity and specificity. Its feature-based nature, however, could also be considered a drawback. As an example, it depends on the quality of API call documentation for the action filter. Furthermore, it must deal with the disparity in syntactic and semantic representations across different programming languages. Moreover, although DeepNN has shown applicability in learning feature vectors, it may not be the optimal choice, and further research is needed to clarify its suitability.

3.2 Self-Supervised Learning

InferCode is based on self-supervised learning principles. Limited effort has been invested in using self-supervised learning on source code, but one notable work attempted using self-supervised learning for program repair [78]. Semi-supervised learning can be mistakenly considered as a self-learning technique, but it indeed it is different as it relies on a portion of labeled data and unlabeled data while self-learning uses the structure of unlabeled data. InferCode represents the source model, which is frozen, and in this work, just what was added the end of the model using Siamese model was trained. Fig. 3.1 shows the steps of transferring InferCode to our current task.

Models such as Convolutional Neural Networks (CNNs) were extensively used in the era before Self-Supervised Learning (SSL) for tasks such as object detection and image recognition. Traditional methods relied heavily on large quantities of labeled data in order to perform well, which presented a significant challenge due to the time-consuming and costly nature of data labeling. In order to alleviate this problem, SSL was initially developed. To reduce the dependency on manual labeling, the concept relied on exploiting the inherent structure of the data to generate labels automatically. An innovative approach used image rotation

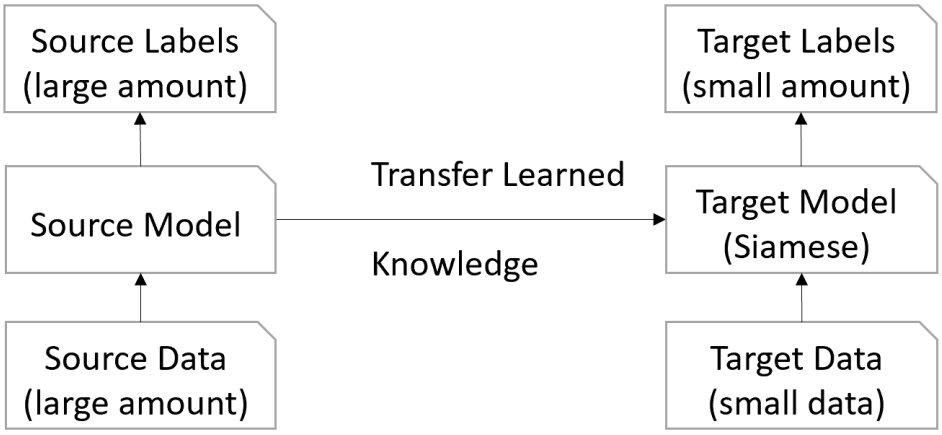


Figure 3.1: Transfer InferCode steps.

as a pretext to generate labels for CNNs [28]. Images were rotated by applying certain rotations, and the model was trained to predict the degree of rotation applied to the original image. As seen in Fig. 3.2, this technique allowed for the creation of a wealth of labeled data without necessitating extensive manual intervention. This form of SSL enables the extraction of useful representations from unlabeled data by using contextual information contained within the images themselves. Its use marked an important step forward in the development of machine learning models, showing that valuable insights can be drawn from data itself, without explicit human annotations. In this way, a new wave of approaches was able to effectively learn from unlabeled or loosely labeled data, greatly expanding the range of feasible applications.

As evidenced in the InferCode tool as in Fig 3.3 success of SSL in image processing sets a precedent and motivates similar techniques to be implemented in other areas such as source code. With SSL, the dependency on human effort for labeling is minimized, making it an attractive strategy for the development of machine learning models across various applications. Nevertheless, it is important to note that the transition of SSL techniques from one domain, such as image processing, to another, such as code analysis, is not trivial due to the

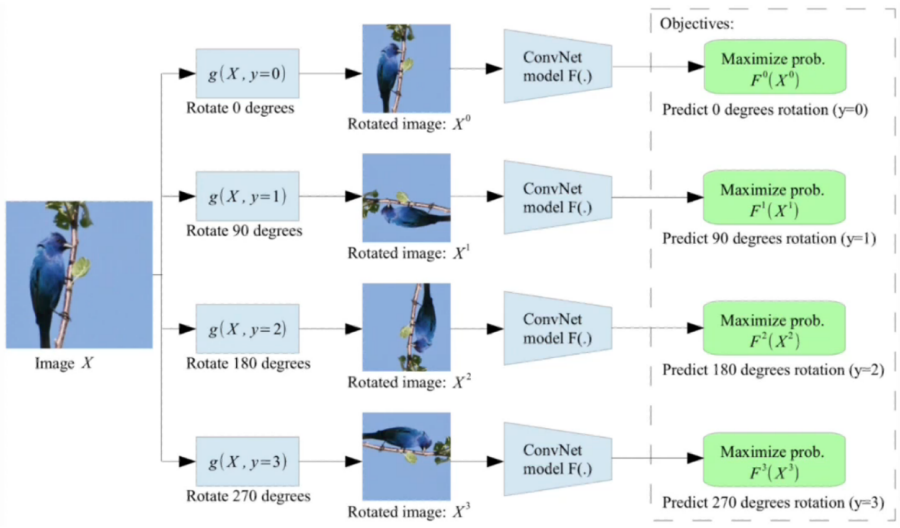


Figure 3.2: The entire input images are rotated to illustrate self-supervised learning. Predicting which rotation to apply is the goal of the model [28].

inherent differences in the types of data and the structures of the various data sets. SSL is being used successfully for the analysis of code, as demonstrated by InferCode, which demonstrates the adaptability and potential of SSL when dealing with a wide variety of data types.

Early approaches in Natural Language Processing (NLP) that uses SSL is Doc2Vec to learn the vector of a document of a paragraph [49], which uses the paragraph ID to predict which word appeared in which paragraph. Doc2Vec comes with 2 version, one that add a paragraph vector to the context of a word is in Fig. 3.4(a), which is distributed memory model for paragraph vectors (PV-DM), that is similar to Word2Vec. The second version is distributed bag of words for paragraph vectors (PV-DBOW) in 3.4(b).

Another one is the masked-language model (MLM) [66]. In such a training scheme, a masked word's representation is learned based on the words that appear on its left and right sides, which makes the model a bi-directional mode. Usually, 15% of the words under MLM are masked before fed to the model. The model is then asked to predict these masked words. As a problem statement, this can be visualized as a fill-in-the-blanks scenario as in Fig. 3.5

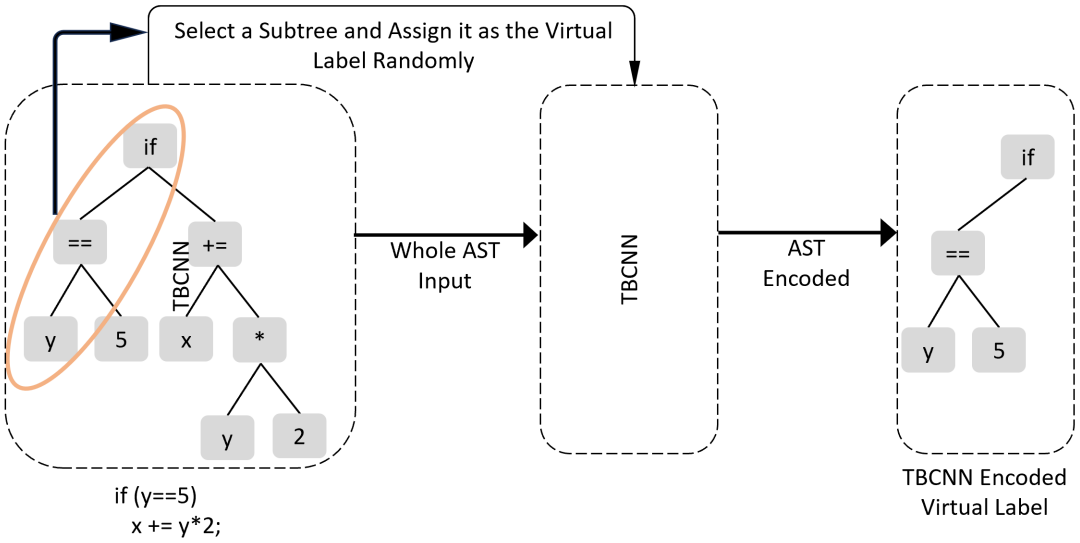


Figure 3.3: Comparison of InferCode's Self-Supervised Learning approach with traditional image rotation SSL. The image rotation method predicts the applied rotation degree in image processing tasks. In contrast, InferCode's approach, while maintaining the self-supervision principle, involves randomly selecting subtrees from abstract syntax trees during training in the context of code analysis. This highlights the adaptability of self-supervised learning techniques across diverse domains.

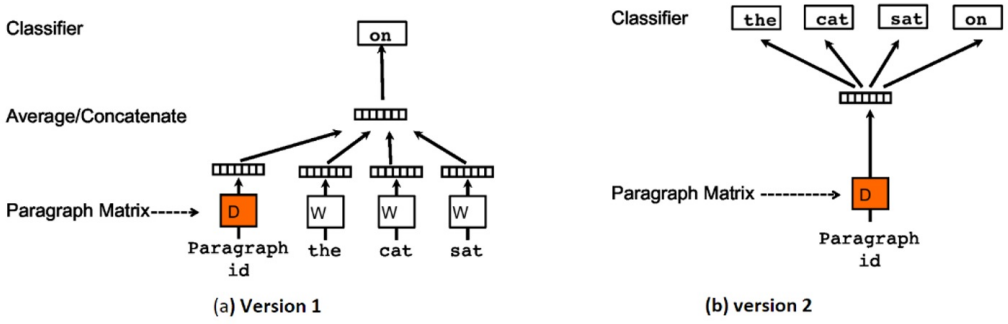


Figure 3.4: Doc2Vec versions [49]. Version 1 is PV-DM and versions 2 represents PV-DBOW

On the other hand, casual modeling language (CML) operates only on the left side of the masked word making then a unidirectional models. Bidirectional Encoder Representation

Figure 3.5: Input to a masked-language model, where the model tries to predict the masked words.

from Transformers (BERT) [24] in Fig. 3.6 and GPT [61] are SLS and MLM-based models that take unlabeled data and generates labels.

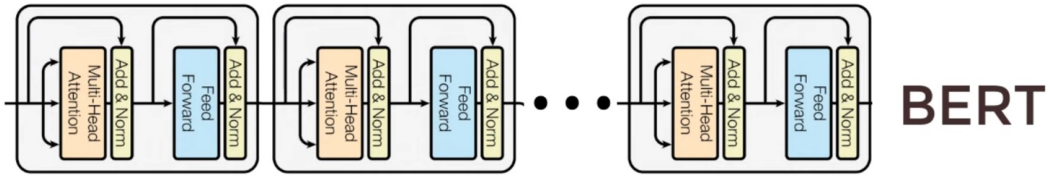


Figure 3.6: BERT representing a stack of Encoders [24].

Before InferCode, there is no SSL-based model for source code. For example, GREAT global relation models for source code [33], GGNN for representing code with graphs [4], and Code2Vec and Code2Seq [5, 6] are all label-hungry models. As a result, millions of source code samples need to be labeled.

3.3 Sequence-to-Sequence (Seq2Seq) Model for Code Clone Detection

Sequence-to-sequence (Seq2Seq) models also known as the Encoder-Decoder model can be used to solve a variety of problems like machine translation, chatbot, text summarization, and question answering [26, 41, 44, 75, 78, 82]. It has also applications in other

domains like cybersecurity for anomaly detection [14]. Seq2Seq is designed to deal with varying lengths of input and output sequences. Seq2Seq takes a sequence of vectors and produces a sequence of vectors as outputs. Sequ2Seq models can be trained for sequence-to-label, label-sequence, and sequence-to-sequence training. To predict the next target output, the Seq2Seq model can also be trained in an unsupervised manner, which is called an autoregressive model. Moreover, a Seq2Seq can be either a casual or non-casual model, where casual can only look backward in the sequence for the current output its producing.

Zorin et al. tested a pretrained RNN Seq2Seq is an AST-based model. They splits code methods into tokens, where some uninteresting AST nodes are deleted [83]. Tokens fed into their model are represented using the Word2Vec model. A mutation strategy was used to create their own dataset of clones, making their work unique, though the trained models were not accurate. Source code word embeddings are fed to RNN Seq2Seq to generate fixed dimensional vector representations for the source code, since word embeddings can vary in dimension. It is then fed into a Siamese network for distance comparison before being decoded by second LSTM decoder and mapped to AST tokens for human interpretation.

3.4 Seq2Seq Model with Attention

Code2Seq and Code2Vec [5, 6] are AST-based, Seq2Seq source code embedding with attention. By predicting method names as in Fig.3.7, these models explain what the program does.

Code2Seq and Code2Vec loop over all possible paths between leaves in an AST tree and create paths between each of 2 leaves in the tree. Then, they find the path that contribute the most to a method name. They train the model with pairs of programs along with the program property $(P_1, \alpha_1) \cdots (P_N, \alpha_N)$, where they hope to predict unseen program P property α, such as method name. Even though AST paths is purely syntactic as it's based on tokens extracted from an AST of a source code, it indeed captures some semantic. The program is represented as a set of AST paths vectors. AST paths vectors are divided into two sets:

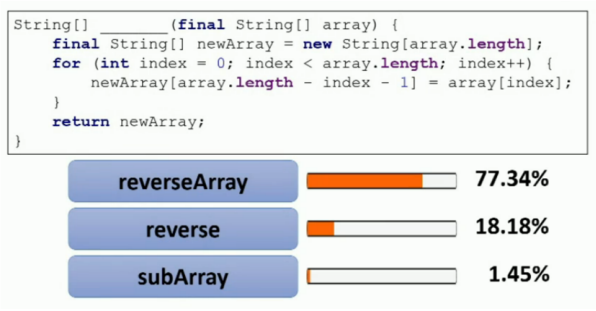


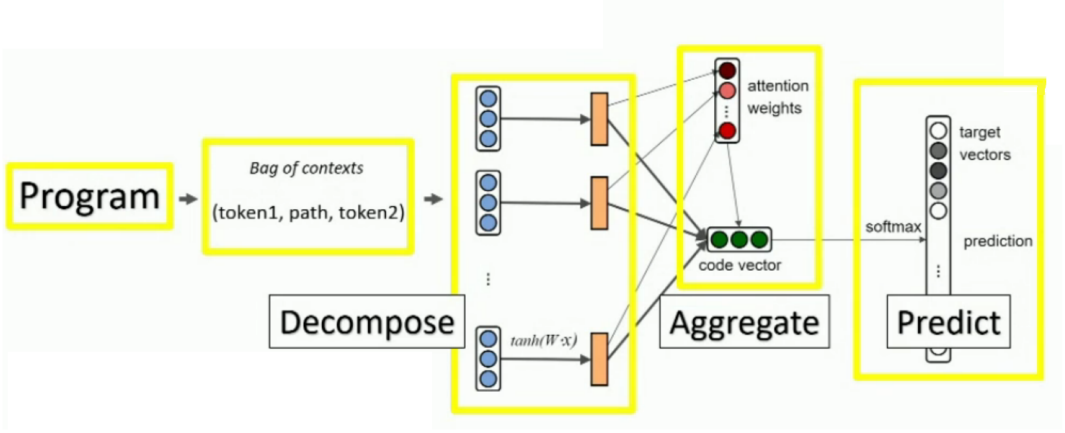
Figure 3.7: Code2Seq model predicts method names. [5].

token vectors and path vectors. First, a token vector represents the tokenization of the two leaves that begin and end the path, where each leaf will have its own vector. Second, the path vector that is a tokenization of the path between the 2 leaves. Finally, they concatenate these 3 vectors for the 2 leaves and the path vector to produce a path context. The path context captures the whole information of the path and the 2 tokens the path connects. The weight matrix W below is learned during training so that it combines them and is updated to learn the whole context of the 2 leaves and the path vector.

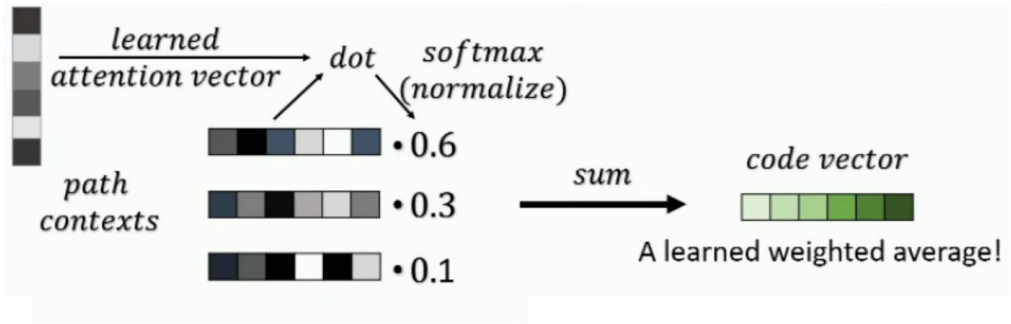
$$tanh(W.[leaf1_{embed}, pathVector_{embed}, leaf2_{embed}]) \tag{3.3}$$

There are many path context vectors as many paths exist. Those path context vectors are of arbitrary size. Since there exist many path context vectors per a program, those path contexts need to be aggregated.. Fig. 3.8(a) summarizes the architecture of the model. Many aggregation strategies are out there such as selecting the most most important path context vector, which could impact the accuracy as we lose information from other path context vectors for the program. Another one is averaging all path contexts for a program, which has the drawback of giving equal attention to informative and non-informative vectors alike. Finally approach is by using attention mechanism by learning the weighted average of path context vectors. The attention mechanism gives the best results as it considers the important path context

vectors. As a result of learning attention weights, they are able to gain semantic insight from the context of a path and also determine how much attention should be given to a path context vector. To achieve the former, an attention vector is learned and is pair-wise multiplied by with each path context vector in a program Fig. 3.8(b).



(a) Code2Vec architecture.



(b) Multiplying attention vector with path context vectors.

Figure 3.8: Code2Vec architecture. [5].

CHAPTER 4

APPROACH

4.1 Data Processing and Data Augmentation

Many benchmarking datasets have been developed to gauge the accuracy of tools like BigCloneBench [72], which contains interproject and inter-project clones of the four primary clone types. The problem with this dataset is that it has pair of clones for one language not across languages.

Part of transfer learning InferCode is to customize it for our downstream task. InferCode embeddings is augmented with label 1 and 0. Label 1 refers to a pair of clones while label 0 refers to disclones. This will help training the Siamese architecture explained in Section 4.4.

Experiments were conducted based on data collected from a programming competition website where submissions are classified per problem title. Each submission should be submitted in only one language either Python or Java per problem title and thus, it can ensure that those submitted for the same title are clones. In this work, submissions for the Fibonacci series problem were collected and paired in Java and Python. In the experiments, the dataset was splitted into 70% for training and 30% for validation. Fig. 4.1 shows the embedding process using InferCode. To help contrastice learning model learns, the collected pairs of Java and Python programs were augmented and labeled by clones and discolnes where clones are labeled with 1 and disclones are labeled with 0. The labeled and unlabeled pairs are then passed to InferCode to produce embeddings.

In order to create unique pairs of Java and Python code, the two datasets of Python and Java are merged using the inner join operation where each row represents a submission. The rows of the datasets are joined if they have the same problem title in the reference column. That is, they are solutions for the same problem. For rows in one dataset and rows in

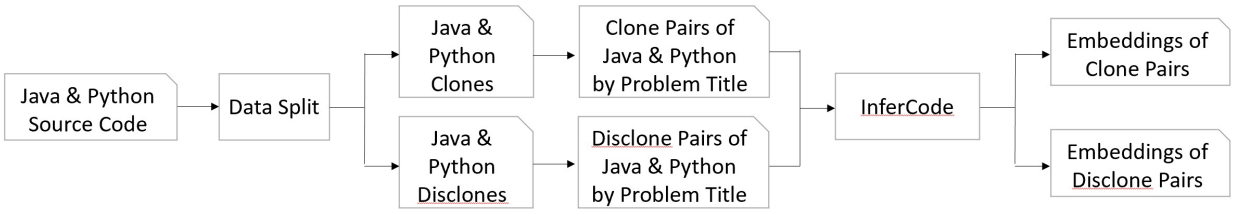


Figure 4.1: Source code embedding using InferCode.

the other dataset for the same problem title, $m \prod n$ merged rows are produced, which is too many. In order to reduce the number of merged rows, another column was added to index the submissions for the same problem title and used it together with the problem title column for merging. Fig. 4.2 shows how the datasets both indexed and unindexed were inner-joined. This results in only $m + n$ merged rows. Algorithm 1 describes the process of creating clone pairs. Disclone pairs are created in a similar process where different problem titles are used.

Algorithm 1 Create Clone Pairs

1: $createClonePairs(dataset) : clonePairs$
2: $< dataset\_Python, dataset\_Java > \leftarrow split(dataset)$ ▷ Split $dataset$ into Python and Java datasets
3: $grouped\_Python \leftarrow groupByProblemTitle(dataset\_Python)$
4: $grouped\_Java \leftarrow groupByProblemTitle(dataset\_Java)$
5: $indexed\_Python \leftarrow index(grouped\_Python)$
6: $indexed\_Java \leftarrow index(grouped\_Java)$
7: $clonePairs \leftarrow join(indexed\_Python, indexed\_Java)$

CodePython	Problem Title
x1	p1
x2	p1
x3	p2
X4	p2
...	...

CodeJava	Problem Title
j1	p1
j2	p1
j3	p2
j4	p2
...	...

CodePython	Problem Title	CodeJava
X1	p1	j1
x1	p1	j2
x2	p1	j1
x2	p1	j2
x3	p2	j3
x3	p2	j4
x4	p2	j3
...	...	...

(a) Inner join without Indexing

CodePython	ProblemTitle	Index
x1	p1	1
x2	p1	2
x3	p2	1
x4	p2	2
...	...	...

CodeJava	Problem Title	Index
j1	p1	1
j2	p1	2
j3	p2	1
j4	p2	2
...	...	...

CodePython	Problem Title	CodeJava	Index
x1	p1	j1	1
x2	p1	j2	2
x3	p2	j3	1
x4	p2	j4	2
...	...	...	...

(b) Inner join with indexing

Figure 4.2: Indexing dataset.

4.2 Case Study: Apply CLCD-I on a Whole Software

4.2.1 KD-tree with Database

A tool that can detect clones in a system faces the challenge of locating the clone given that there are thousands of lines of code within the system. Several popular clone detection techniques have shortcomings, such as not supporting Type-3 near-miss clones where lines can be modified, added, or deleted, and not scaling adequately to detect clones in large systems [38]. There is also the challenge of retrieving and querying possible clones. While fingerprints or hashes can be helpful for very similar clone pairs, they can be difficult for in-

exact matches. The Simhash algorithm detected near duplicate documents over millions of pages with high efficiency [18]. Is it possible to apply it on Type III we studied using InferCode given a large codebase? To evaluate the potential tool, it can be used against Nicad. [20] clone detection since it can also detect Type III clones, which is the main type the tool targets.

Given a whole software that has source code for Java and Python, is it possible to apply the tool CLCD-I on the system to find out clones between methods? The problem of finding clones can be addressed in the perspective of data mining. The hashing algorithm Simhash in Simhash-based NiCad algorithm [73] can be replaced with CLCD-based NiCad and compare both of them. However, that is not a fair comparison since Simhash-based NiCad works differently and the output of the algorithm is different from the output of InferCode. Alternatively, the use of clustering algorithms of embeddings is suggested.

One approach is to embed the 100 dimensional vector for a method in a system to 3 dimensional using Principal Component Analysis (PCA) [1], but that could cause a loss of information from InferCode embeddings. The first run on the entire system needs to be carried out, which is expensive. The drawback of this strategy is that k for k-means algorithms cannot be determined. There may be several categories of code that do different things, so even if the selection of k is dynamic, k might grow.

A nearest neighbor search algorithm can be used to find similar source code based on the embeddings of the source code of the system. A database of all embeddings can be searched for the nearest neighbors of an input source code given its embedding. An efficient way to implement a nearest neighbor search algorithm is to use a data structure like a KD-tree or a ball tree. This approach has the advantage of not requiring you to predefine the number of classes or clusters. Based on the similarity of their embeddings, it simply finds the nearest neighbors. Several distance metrics (such as Euclidean distance, cosine similarity) can be used with this approach, including large datasets and high-dimensional embeddings. There is also a deep learning-based approach, such as a convolutional neural network (CNN) or a siamese network. An unsupervised or supervised approach can be used to train these mod-

els to map codebase to embeddings. Using the learned embeddings of a CNN, the similarity between pairs of codebase can be predicted and a nearest neighbor search using the learned embeddings can be performed.

Whenever a new method is added to the system, the KD-tree is queried first to see if it has clones. The method embedding is added to the system if there are no potential clones. Given an input method embedding of dimension 100, neighbors and potential clones can be found, by searching the KD-tree. As the KD-tree is kept down, all potential clones can be retrieved by retrieving a node in the tree representing an embedding for a method if the Euclidean distance is greater than a threshold 0.7. This threshold is set to be consistent with the base work in this work.

Nevertheless, InferCode gives 100 dimensions, which is quite high for KD-tree, which does not work well for high-dimensional data. In a KD-tree, each dimension is indexed at a different level, so when a query is made, the algorithm will perform a lot of backtracking (searching both sides of a branch) and end up searching most points. A tree structure loses its advantages when this happens and an exhaustive comparison is faster than a tree structure. Once any method is added to the system, the instance is immediately added to the clone detection system.

4.2.2 k-NN with Faiss Index

Faiss is a vector database. This is where you store vector representations of text, images, or whatever data you have (e.g. embeddings). Based on another query vector, a search can be done by vector similarity. For instance, search engines can be implemented based on similarity. The difference between a Fiass index and a database index should be clarified as follows: database indexes are very similar to those found in books. However, the Faiss library used in this case study is based on an encoding of the InferCode vectors that is used to search for similarity between Python and Java sourcecode. On the other hand, traditional database indexation is used for exact lookups in the database.

The purpose of this algorithm is to efficiently find the most similar high-dimension vector from the input vector. Typically for machine learning purposes. A perfect match is not mandatory.

In brief, Faiss works as follows:

- Vectors are encoded with the help of vector encoding. In the case where each dimension is stored in its own field or column, we bring them all together into one field and store them together as one. There is even the possibility of a reduction in dimensionality, but this must be balanced against any loss of information that may occur.
- Creating the index is the first step in the process. During this process, various methods can be used to index the encoded vector in a manner comparable to that used in traditional databases.
- The querying process. In contrast to traditional database lookups that look for exact matches within the index, FAISS relies on a similarity metric (e.g. cosine similarity) to find the best possible match within the database.

Below a potential system design is demonstrated to find clones in a large-scale software in Fig. 4.3. The purpose of InferCode is to transfer the source code input for either a Python or a Java into a 100 dimensional embedding based on AST representation, which is a useful representation for Type III clone detection.

If two source codes do searching functionality in Java, Python, or the same programming language, their vectors would be very close. There is, however, a considerable distance between a method for sorting and a method for calculating stock market price. K-NN index is the second part of the clone detection system. Upon receiving an input source code representation based on the InferCode, the recommendation algorithm computes the corresponding query vector. Based on the query vector, the K-NN algorithm will find the most similar vectors.

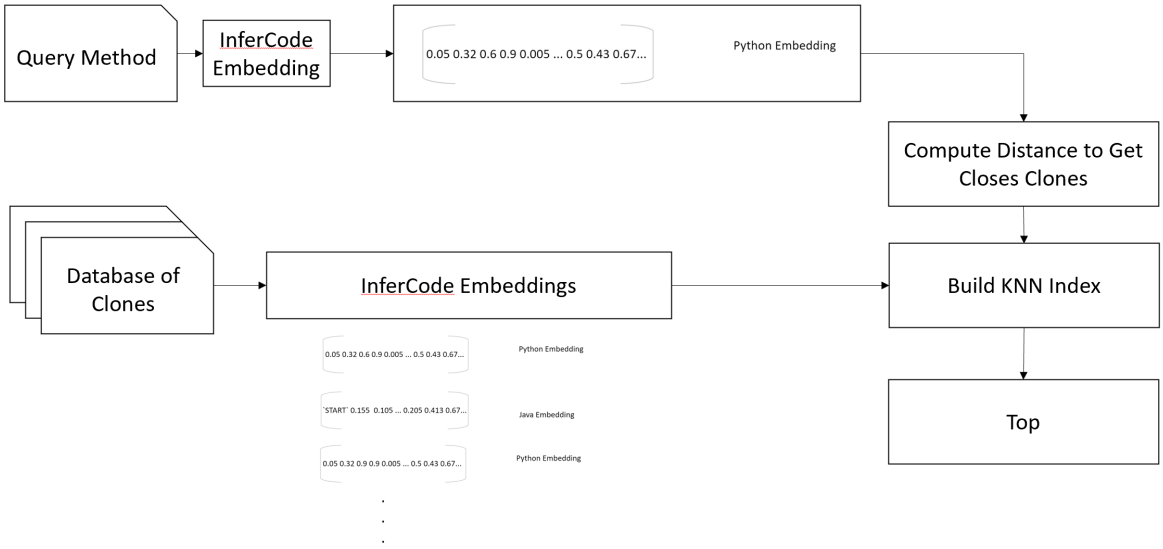


Figure 4.3: Indexing dataset.

The K-NN index can be built using Faiss tool in Python. IndexFlatL2 works very similarly to how K-NN algorithm works as it selects k closest neighbors to a given source code embedding. Index is first built, then InferCode embeddings are passed along with the target embedding for the source code where clones are to be found and it will return result among millions of code samples InferCode representations. An option is given to return the IDs of the returned clones for the target clone where they can be inner-joined with their original source code representation originally stored in the database along with their embeddings. One disadvantage to IndexFlatL2 is that it takes more time than other indexes in Faiss, but it has high quality though. Another index from Faiss is Locality Sensitive Hashing (LSH) which works by increasing collisions on the target buckets instead of avoiding them, which turns out indeed to help cluster the data into similar groups. If for a given input query source code, the bucket empty is found, the hamming distance is used to find the nearest bucket.

4.3 Generating Embeddings with InferCode

In transfer learning, the model can be customized to solve several downstream tasks with the help of a dataset. After the date processing step in Section 4.1, embeddings is aug-

mented with labels 1 and 0 to indicate clones and disclones, respectively. As a result, Siamese model can be taught what clones and disclones to be found during training of Siamese architecture.

InferCode trained Tree-Based Convolutional Neural Network (TBCNN) to encode source code into an embedding [55]. This pre-trained model then can be used for several downstream tasks, including our extension for cross-language clone detection.

InferCode itself is not fine-tuned but the model is transfered to solve the problem. InferCode trainable parameters are freezed as only the output layer of InferCode is of interest. It's recommended that learning rate for transferred model to be slightly higher than a fine-tuned model [13]. To that end, different configurations of learning rate were tried to find the one that gives the best result. The output of the transfer learning model in this work's case is another model, which is Siamese architecture. Siamese architecture will learn distance between 2 embeddings produced by the frozen model, which is InferCode.

Since we have high quality features from InferCode, we set the learning rate and the number of epochs to be relatively small as we have a lot of information already stored in the features given by InferCode. Fig. 4.4 shows how InferCode can be used for several downstream tasks.

4.3.1 Pseudo-labelled data and pretext Task Protocol and Self-Supervised Learning

It was challenging in InferCode to produce pseudo-labeled data from source code for the encoder. The InferCode training consists of the following steps:

- Train an encoder to give a data representation using cross entropy loss function.
- The labelled data is not human-based but pseudo-based, such as rotations for computer vision.

The problem in this work is supervised contrastive learning, where the distance between close pairs (clones) should be minmized and the distance between far pairs (disclones) should be maximized. InferCode performed the pretext task for the project code in this work,

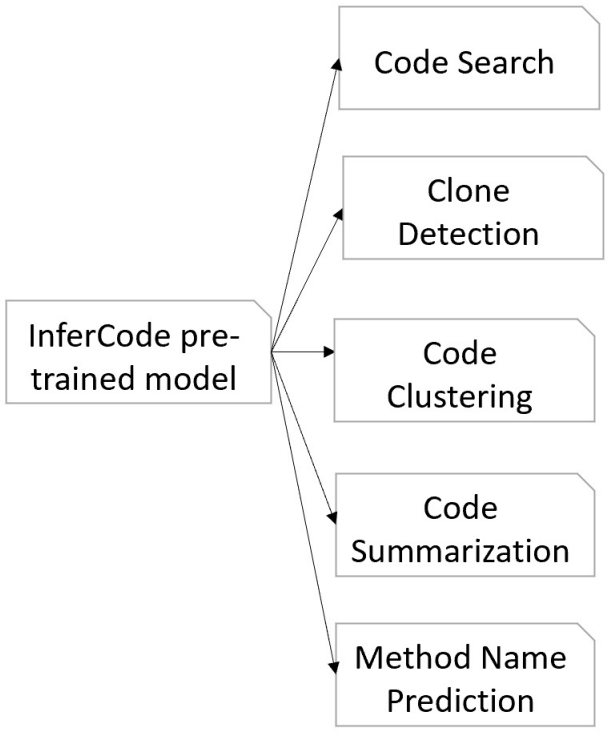


Figure 4.4: InferCode several downstream tasks.

where it predicted a subtree for a source code AST given as input to InferCode. As we stated in Section 2.1, Type III clones have similar subtrees. AST trees are more likely to contain Type III clones at subtree levels since we are targeting Type III clones. Therefore, InferCode appears to be a perfect choice when it comes to pretex model tasks. Using Python and Java source codes, it can be understood whether the predicted subtrees for both Python and Java source codes are sufficient to predict clones of Type III when given two source codes. The model predicts whether unseen subtrees belong to certain trees after it has been trained. Using Siamese networks, their knowledge is applied to the downstream task described in Section 4.4. Rather than adding an output layer and classification layer to InferCode's pretex model, a whole new network is added.

As a pretext model for learning the AST representation of a source code in supervised contrastive SSL, the InferCode is used to learn AST representations using pseudo-labeles,

which are subtrees extracted from AST for a given source code, thereby removing the need for human interaction in data labeling. As in fully supervised, only the limited labeled data can be learned, and it is not allowed to learn the unlabeled data, which is the source code in the problem. InferCode models 'observe' and 'learn' feature representations of source code from all source code repositories regardless of their labeling. It is similar to a baby learning image of all animals and being able to identify them without knowing their names. The downstream task can be achieved using transfer-learning the trained or fine-tuning the SSL InferCode model on the limited data in our problem. Since the kid is already familiar with all the animals, the kid knows, for example, this is an elephant and that is a tiger. The kid knows feature representations from all animals and can quickly, by using a few examples in a downstream, learn that elephants have large ears and a long proboscis. That is what was done to InferCode model. More labelled data to learn more details about clones.

As explained at the beginning of Section 4.3, the InferCode policy for defining the pretext task is useful for many downstream tasks. InferCode's downstream tasks can be extended to include detecting cross-language code clones. It should also be noted that if it is known that water and cement are significant contributors to the strength of concrete mix, then a pretext that augments cement and water rather than other mix components can be constructed. If it is known that out of the 16 EEG channels, on average, the signals from channels 6, 7, and 15 are highly influential in addressing early epilepsy recognition, a pretext task by augmenting the signals from these channels rather than other channels can be designed.

4.3.2 An intuitive understanding of InferCode

In the 2 programs for loops below, it can be seen the initialization of the for loop in the 2 programs in Java and python are similar as in Fig. 4.5. It shows similar statements as well in swapping elements of the array. By leveraging similar parts of the programs, InferCode produces pseudo-labels. Similar programs that have similar statements should be in the same vector space.

```java
public void selectionSort(int data[]) {
   for (int i = 0; i < data.length - 1; i++) {
     int min = i;
     for (int j = i + 1; j < data.length; j++) {
       if (data[j] < data[min]) {
         min = j;
       }
     }
     int temp = data[i];
     data[i] = data[min];
     data[min] = temp;
   }
 }
```

(a) Selection Sort in Java

```python
def selectionSort(data, size):
    for i in range(size):
        min = i
        for j in range(i + 1, size):
            if data[j] < data[min]:
                min = j
        (data[i], data[min]) = (data[min], data[i])
```

(b) Selection Sort in Python

Figure 4.5: Java and Python selection sort programs.

The concept is similar to Doc2Vec in Fig. , where words are treated as pseudo-labels. Let it match how Doc2Vec works to InferCode. Each subtree (corresponding to a word in the text domain) is regarded as a pseudo-label for a source code (corresponding to a document in the text domain). The subtree in InferCode is sampled randomly from an AST for a program. After InferCode is trained, the encoder will map any source code into a vector representation to that pseudo-label subtree. This new pseudo-label subtree can be used for comparison for another source code subtree. This is how similar program statements from different languages could be compared for clone similarity.

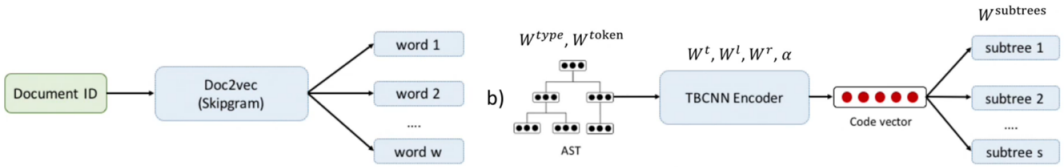


Figure 4.6: Comparing InferCode to Doc2Vec.

After InferCode is trained, the encoder will map any source code into a vector that should be similar to another program with similar statements. The steps involved in generating pseudo-label data for InferCode are as follows:

- Traverse source code AST.
- Selecting subtrees that are not large.
- Large subtrees are unique to a program, thus dropped.
- Select 5 grand subtrees.

In this work, instead of fine-tuning the Infercode pretrained encoder in Fig. 4.7, the code vectors was used instead. Infercode encoder can be used for various downstream tasks like clone detection, but it is extended to cross-language clone detection.

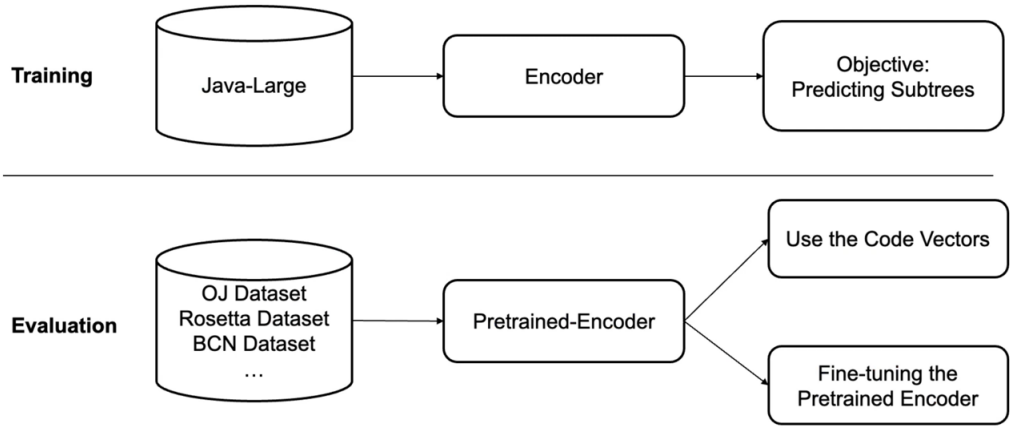


Figure 4.7: Training and evaluation of InferCode as shown by the authors of InferCode [16].

4.4 Siamese Architecture

Instead of transferring and fine-tuning InferCode and train it with the labeled data in this work, the embeddings generated by InferCode was used and the model developed in this work was trained on top of it. The model presented in this work is called Cross Language Clone Detection with InferCode (CLCD-I). A Siamese architecture, a contrastive learning model is used, for comparative process of Java and Python embeddings to determine their cloneness. Fig. 4.8 shows the process of CLCD-I. The collection of Java and Python code pairs is split into clones and disclones pairs. The split pairs are then input to InferCode to generate embeddings. The embeddings are fed into a Siamese architecture for comparative process of Java and Python embeddings.

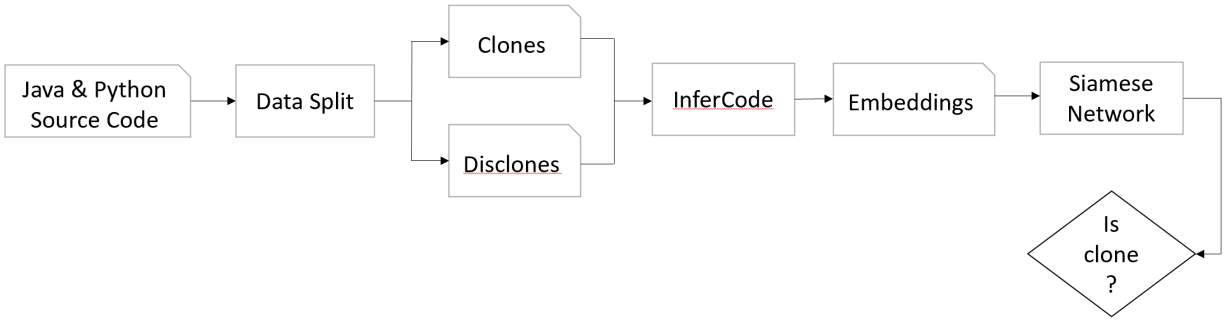


Figure 4.8: CLCD-I model architecture.

CHAPTER 5

VALIDATION

5.1 LSTM Models

5.1.1 LSTM unit

LSTM unlike other models uses one LSTM unit to for all input, where it takes one item of the input at a time and gives output. It does not update weights and biases while unrolling itself and read the input in order to adapt itself with data sequences of different lengths. In this work, 16 LSTM units were used. In Fig. 5.1 the layer is replaced with 16 to have a complete picture of LSTM autoencoder used in this work.

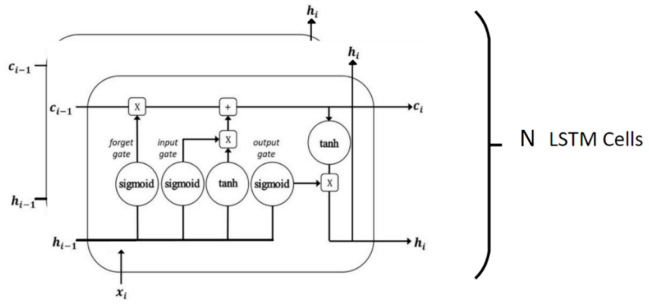


Figure 5.1: CLCD-I model architecture.

Any LSTM cell has 3 return types: return last hidden state and cell state, return all hidden states or return all hidden states including last cell and hidden state. In this work, all hidden states were used including cell state. This is enabled by setting *returnState* and *returnSequence* both to true.

5.1.2 LSTM Encoder Decoder

The problem being addressed using LSTM Encoder-Decoder is translating a Python embedding into Java. The embeddings below represent what InferCode summarizes about AST of both Java and Python in Fig. 5.2.

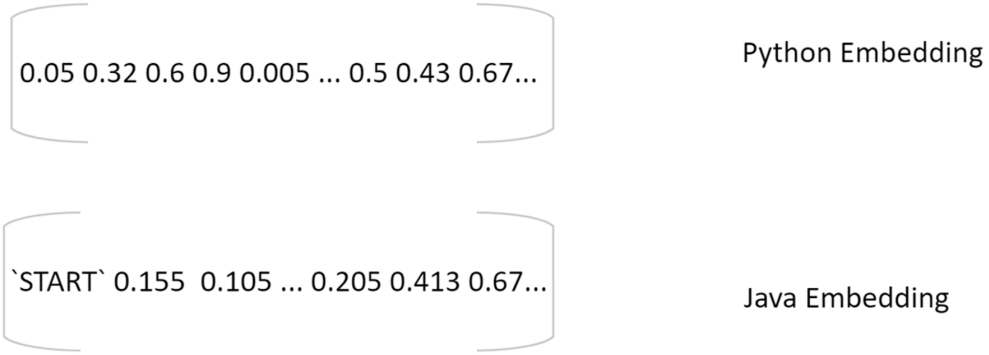


Figure 5.2: Python Embedding to Java Embedding from InferCode.

Encoder-Decoder through LSTM cells are used as shown in Fig 5.3. The previous section discussed that each LSTM cell outputs cell state and hidden states. The cell state is meant to encode a kind of aggregation of data from all previous time-steps that have been processed, while the hidden state is meant to encode a kind of characterization of the previous time-step's data. Introducing the cell state into the LSTM cells actually increased the complexity of the model. As a result, increasing complexity will usually increase variance and decrease bias. The cell state acts as a highway in order for the gradient to flow better to the earlier states, which in turn allows the model to capture memory that are further back in the past.

In test cases, the number of tokens/timesteps is defined as the number of inputs per timestep taken from the embedding of Java and Python. The input was divided into four parts

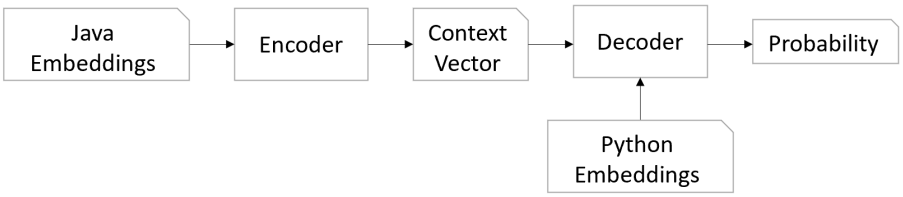


Figure 5.3: LSTM Encoder-Decoder.

of 25 length each. This means that there are a total of 4 timesteps. Then features of each of the 4 parts are defined to be 25 taken from the embedding of Java and Python. InferCode embedding is used to represent both Python and Java code.

The encoder encodes the Python/Java input sequence into a new representation, the so-called context vector that represents a summary of the input sequence. In other words, this represents a summarization of encoded input. Decoder decodes the context vector into output sequence as shown in Fig. 5.4. Context vector is the last hidden and cell state of LSTM encoder, which will be the initial hidden and cell state of the Decoder.

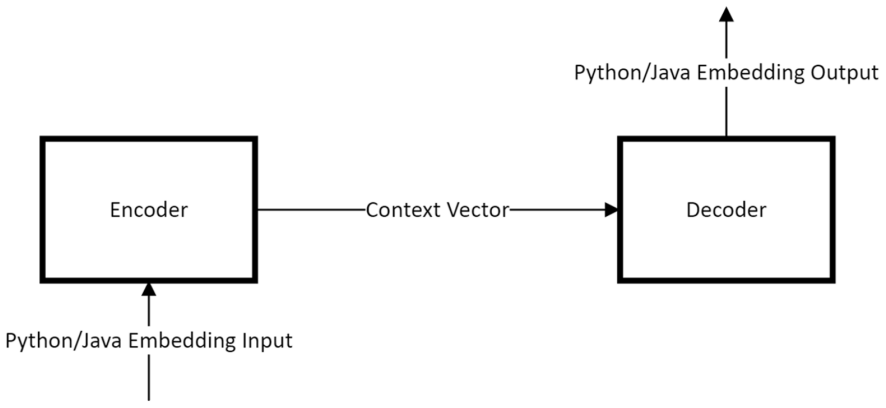


Figure 5.4: LSTM Encoder-Decoder.

Encoders and Decoders are trained to work together to create a context (representation) between input and output. In prediction phase, they will continue to work together to predict the output. Decoder decode the context vector and predicts output one at a time. At the beginning of training, the input is given and then the LSTM hidden state and cell state are provided and the last hidden state is dropped. Since return state is set to true, it will return the last 2 hidden states as well as the cell state (c_i, h_i, h_i) as Fig. 5.5 shows. Only 2 of them (c_i, h_i) were used because each LSTM cell state uses 2 values to initialize its initial states (c_{i-1}, h_{i-1}).

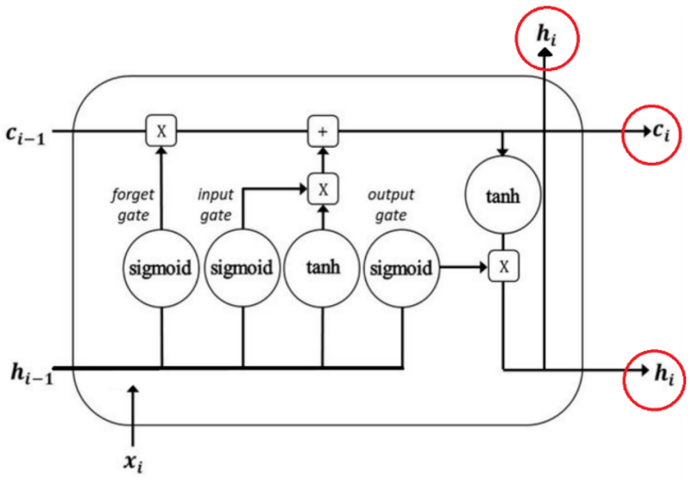


Figure 5.5: What LSTM cell returns when we set return state to true.

After the decoder LSTM is initialized with encoder internal LSTM state (context vector). Then it will begin creating the output by taking '[Start]' and then output something and then using that something, it will create the next thing until all embedding input of Python '[START] 0.05 0.12 . . . ' are consumed until '[STOP]' is produced as in Fig. 5.6.

Back to the context vector, the dimension is defined based on number of LSTM units we have. 16 LSTM cells are stacked on top of each other, where each LSTM cell returns 3

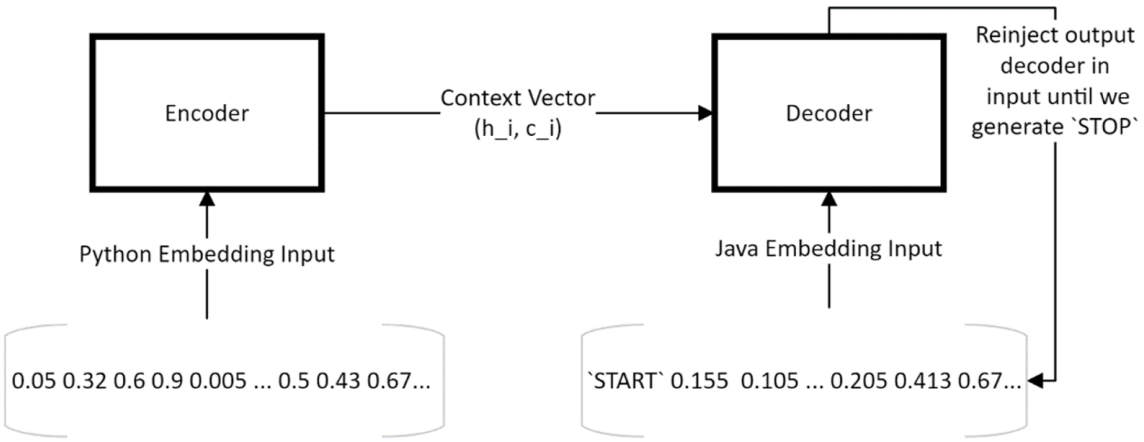


Figure 5.6: Encoder-decoder model information transmission. Internal LSTM states represent context vector.

2D tensors for cell state, hidden state and hidden state again. So, there is 16×3 $\left(h_i, c_i, h_i\right)$ for each cell. Enbedding input sequence is not too long (100 dimensions), so the number of LSTM units is set to 16 as Fig 5.7 shows. As the input vector grows, more condensed information is piled in the context vector, so this is carefully monitored to avoid losing information by having few LSTM unit cells. Increasing the number of layers in the LSTM encoder is another option to avoid losing context vector information. In test cases, a combination of both was used where the number of LSTM units was set to 16 and dropout layers were used between timesteps to avoid overfitting the model. The dimension of each hidden and cell state equals to the LSTM unit number$[None, numberUnits]$. Since the number of units is set to 16, then $[(None, 16), (None, 16), (None, 16)]$.

To summarize, the LSTM Encoder works as follows: return sequence is set to true to return the last hidden state h_i , last hidden state again h_i and last cell state c_i of the last time step. Due to the fact that there are two last hidden states, the first one (actually, the first one is considered the output of the LSTM in general) can be ignored.

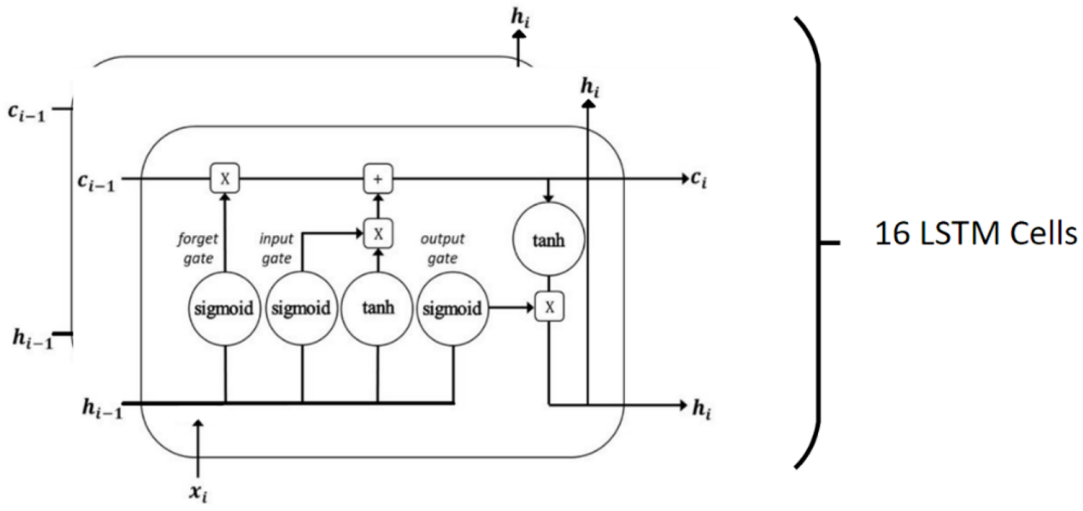


Figure 5.7: Each LSTM cell has $\big(h_i, c_i, h_i\big)$.

For the decoder, the decoder LSTM is defined as follows: context vector from encoder, here $\big[state_h, state_c\big]$ can be used, to initialize the decoder LSTM initial states. Then, a decoder LSTM layer is added and set to return all hidden states, and to return cell states as well by setting return sequences and return state parameters to true. After Decoder output, a dropout layer is added before passing the Decoder output to another Dense layer with a Soft-Max layer that will predict the output. A dense layer is used to predict one timestep (each of dimension 25) at a time. So, one output is produced at a time with LSTM decoder. Both cell state and hidden state have $[(None, 16), (None, 16)]$, which are the output of Decoder are going to be used as the next decoder state. The hidden states usually embed what is has seen so far at each timestep.

To sum up, the Decoder produces the output sequence one by one. For each output, the decoder consumes a context vector and an input. The initial context vector is created by the encoder. The initial input to the decoder is a special symbol for decoder to make it start, e.g., '[zero]' or '[start]'. Using initial context and initial input, the decoder will generate/predict the first output. For the next time step, decoder will use its own last hidden and cell states

as context vector and the generated/predicted output from the decoder at the previous time step as input. Decoder will work in such a loop using its states and output as the next step context vector and input until:

- the generated output is a special symbol (e.g., '[STOP]' or '[END]'), or
- the pre-defined maximum steps (length of output) is reached, which is the case in our experiment.

Fig. 5.8 shows both accuracy and loss results of our LSTM Encoder-Decoder model. The model learns poorly as no convergence is observed at all between train and validation error. The test accuracy of the model is 0.65%, which is slightly the same to the base work reported by CLCDSA [57] and Perez [60], where both achieved 0.66% accuracy.

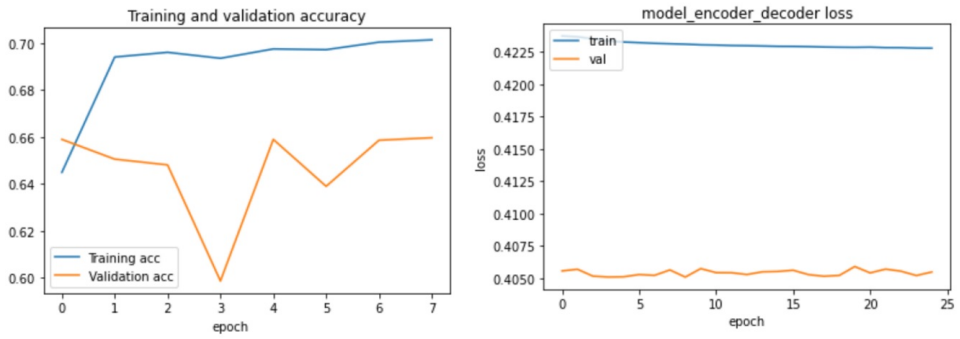


Figure 5.8: LSTM Encoder-Decoder accuracy on the left and loss on the right.

5.1.3 Bahdanau and loung attention

Attention does have its application in many fields of deep learning such as in Computer Vision, and in NLP tasks [27, 31]. As Attention was developed to handle long sequences in machine translation, it is also applicable to most other NLP tasks.

Due to the fact that only the last hidden state of the encoder RNN is used to construct the context vector for the decoder, the standard seq2seq model generally cannot process long input sequences accurately. LSTM encodes the entire input sentence into a context vector, which is the last hidden state of the LSTM/RNN. It is expected that this will be a good summary of the input. During decoding, the Attention Mechanism directly addresses this issue by retaining and using all hidden states of the input sequence. Throughout the decoding process, the Attention Mechanism retains and utilizes all the hidden states of the input sequence. This is achieved by creating a unique mapping between each decoder time step and each encoder hidden state. To produce each output, the decoder has access to the entire input sequence and can selectively select specific elements from it. This mechanism enables the model to focus and place more "Attention" on the appropriate input sequence parts.

Attention started by storing all hidden states in one memory cell, but due to inefficiency as all hidden states from all timesteps need to be stored, an alternative approach is to combine them into one vector. The combination can be done using a simple element-wise addition as in the work of Bahdanao et al. [8] or using element-wise multiplication as in Loung et al. work [52]. Addition or multiplication can be used since the hidden states output from the encoder are of fixed size.

LSTM with Bahdanao (LSTM+BA) and Loung (LSTM+LA) is a sequence-to-sequence attention model. Main rule of attention in LSTM is to manage and quantify the interdependence between input and output, in what is known as general attention, or within the input itself, or the so-called self-attention. Bahdanau et al. reported that Seq2Seq model with attention achieved higher results than traditional Seq2Seq without attention on long sentence sequence. Instead of remembering the entire input sequence, models with attention are able to focus on specific tokens from the input embedding before making a prediction. Additionally, the context vector contains information from all timesteps, whereas the decoder might only need the first few. The solution is to wait for certain encoder vectors more than others before

adding them point-wise. The weights of words that are more important for the decoder output will be higher. Consequently, the context vector will contain the most important words.

It is described how attention can work with Java and Python embeddings similar to how it would work with machine translation problem. LSTM's Attention component maps each word in the output sentence (Python) to the important and relevant elements of the input sentence (Java), assigning higher weights to these tokens, thus improving output prediction accuracy. Among many types of attention to suit various tasks, we will cover the one used in sequence-to-sequence LSTM model.

Fig. 5.9 shows LSTM+BA, which is commonly referred to as additive attention, which involves aligning the decoder with the relevant input sentences and implementing attention mechanisms. The mechanism is implemented as follows: 1) each element in the input sequence has a hidden state produced by the Encoder, 2) each hidden state of the encoder is aligned with the previous hidden state of the decoder to find the alignment score, 3) alignment scores for each encoder hidden state are combined into a single vector and softmaxed, 4) the encoder hidden states and their respective alignment scores are multiplied to form the context vector, 5) the context vector is concatenated with the previous decoder output and fed into the Decoder LSTM for that time step along with the previous decoder hidden state to produce a new output, and 6) throughout the decoding process, the steps (steps 2—5) repeat themselves until an output token or a maximum output length is reached.

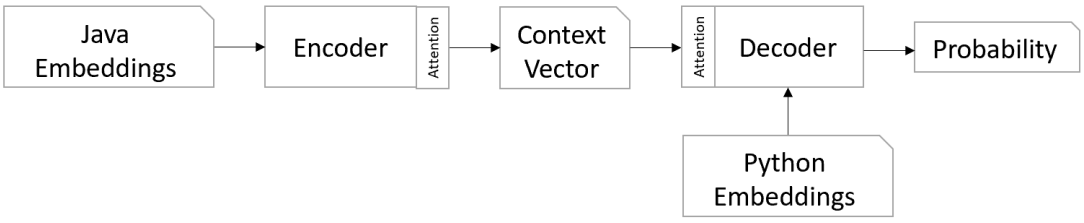


Figure 5.9: LSTM+BA.

5.2 Siamese Architecture

Fig. 5.10 shows a Siamese model used in this work. It consists of two equivalent networks with the same architecture and parameters that can be trained. To learn clones, the model takes similar embeddings as input, and to learn disclones, dissimilar embeddings. The model has alternation of dense and dropout layers for deep learning while reducing overfitting. Unlike Perez et al.'s work [60] where different weights were used between the two networks of their Siamese model, we use the same weight in both networks of our model. They considered that the two networks have different domains as each network takes a different programming language. However, in our work, the domains are of the same nature as both networks take embeddings at the statement-level and thus, the same weight should be used. The weight is updated consistently between the networks during backpropagation. Similarity is measured between the output of the two networks using the pair-wise contrastive loss function [32] which calculates the Euclidean distance between the embeddings in a pair. If their similarity is above the preset threshold [15], then the pair is considered as similar and labeled with 1. Otherwise, the pair is considered as dissimilar and labeled with 0. The output of the loss function is backpropagated for learning. The model is updated only for similar clones to reinforce distinguishment of similar clones from dissimilar clones.

The training set for a Siamese network consists of triplets $(x_1, x_2, y)$where $x_1 \in \mathbb{R}^{100}$ and $x_2 \in \mathbb{R}^{100}$ are sequences of InferCode embeddings and $y \in \{0,1\}$ indicates whether x_1 and x_2 are clones $y = 1$ or disclones $y = 0$. The aim of training is to teach Siamese about similar pairs in embedding space as we don't update the network if the pair is a disclone. The dimension of the source code is fixed to 100 for any source code in a pair (x_1, x_2).

Dataset for training seq2seq models is the same dataset as for Siamese neural network. Vector representations from InferCode are used for a fair comparison for all models used in this work. Optimization of the parameters is done using RMSProp optimization. [30] and loss minimization. For a pair of a Python embedding $py_{emb}$and a Java embedding$java_{emb}$,

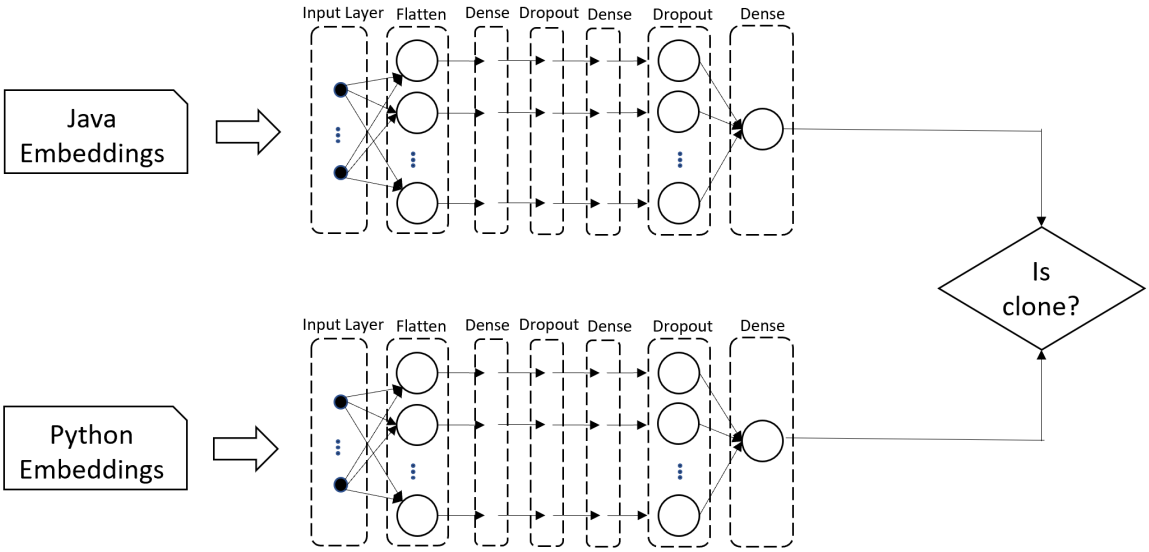


Figure 5.10: Siamese architecture taking as input embeddings of both Java and Python.

the Siamese model produces the weighted output $O_W\left(py_{emb}\right)$ from the Python network and the weighted output $O_W\left(java_{emb}\right)$ of the Java network. Given those outputs, their Euclidean distance approximating the similarity of the embeddings is computed as in Eeq. 5.1.

$$D_W\left(py_{emb}, java_{emb}\right) = \| O_W\left(py_{emb}\right) - O_W\left(java_{emb}\right) \|_2 \tag{5.1}$$

The Siamese network used in this work contains an input layer, 3 dense layers and 2 dropout layers to avoid overfitting the model to our dataset. The output for Python embedding obtained from the final layer is projected to contrastive loss function for comparison against the other input from Java. To learn the input pair similarity, Euclidean based contrastive loss is used. Given the distance, the loss function is defined as Eeq. 5.2.

$$L\left(Y, W, py_{emb}, jv_{emb}\right) = (1-Y)\frac{1}{2}\left(D_W\right)^2 + (Y)\frac{1}{2}\{\max\left(0, m - D_W\right)\}^2 \tag{5.2}$$

where $Y \in \{0,1\}$ (1 indicates similar and 0 indicates dissimilar) and m is a margin to limit the contribution of a dissimilar pair to the loss function. That is, only the pair whose distance is within the margin is allowed to contribute to the loss function. If the distance is higher than the margin $m < D_W(x_i)$, $m - D_W$ becomes negative and thus, 0 is chosen, which makes the distance ignored. That is, for a given pair x_i, the equation results in $(1-Y)\frac{1}{2}\left(D_W(x_i)^2\right) + 0$ if the pair is similar and $0 + (Y)\frac{1}{2}\left(\max\{0, m - D_W(x_i)\}^2\right)$ if dissimilar.

Given Eeq. 5.2, the total loss for K pairs (K similar pairs and K dissimilar pairs) is computed as Eeq. 5.4.

$$L(W) = \sum_{i=1}^{2k} L\left(W, Y, py^i_{emb}, jv^i_{emb}\right) \tag{5.3}$$

Using the Fig. 5.10, we are able to determine how many layers and were used in the experiment. Accuracy of the model and the loss are plotted against number of training examples Fig. 5.11. The accuracy is over test dataset, that's why it stops at 25 epochs while loss improving after 40 epochs, thus it terminates. Fortunately, validation loss and training loss are very close to each other. The following results show that our model is overfitting with learning rate set to 0.001, which is the default in Tensorflow.

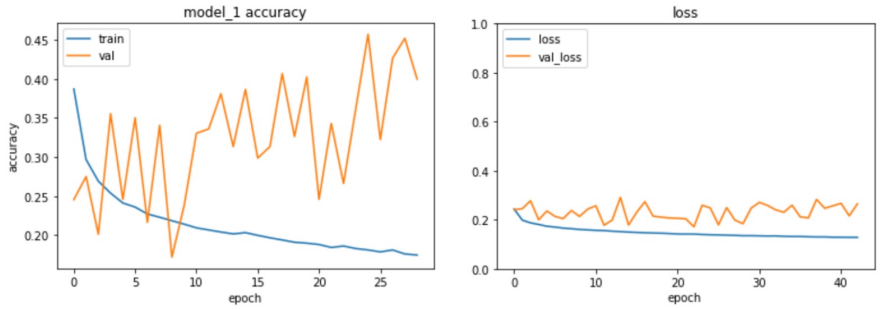


Figure 5.11: Siamese model accuracy and loss with learning rate 0.001.

When the learning rate is set to 0.0001, the obtained results are shown in Fig. 5.12. It now shows better training performance as compared to the validation loss in Fig. 5.11, which showed signs of overfitting. After changing the learning rate, the model trains more stably and is nearly identical after epoch 7 in Fig. 5.12.

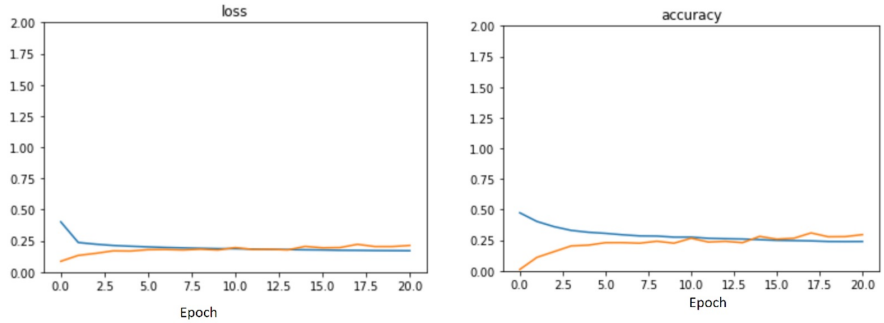


Figure 5.12: Siamese model accuracy and loss with learning rate 0.0001.

5.3 Siamese Variants

It is intended to conduct a series of experiments involving diverse CLCD-I variant implementations in our research. The primary objective of these investigations is to examine the possibility of improving learning from InferCode embeddings by modifiying the architectural dimensions of our model, which can be achieved to enhance learning from InferCode embeddings. This work proposes to expand both the 'width' and 'height' of the Siamese neural network model architecture in terms of their respective widths and heights. As discussed, the 'width' refers to the number of neurons in a layer, and by increasing it, the model's capabilities are enhanced to learn a wide variety of features from the InferCode embeddings. In the same way, the 'height' of the model, which refers to the number of layers included in the model, could be increased in order to allow for more complex and hierarchical representa-

tions of the embeddings, which could lead to a deeper understanding of what the clones are doing. By adjusting the underlying architecture, it is aimed to determine if a more complex network will be able to better harness the rich information embedded in the embeddings of InferCode in a more efficient manner. There will be a crucial consideration in this study concerning the complexity-accuracy trade-off, as our aim is to find an optimal balance that maximizes the model's accuracy in detecting cross-language clones without overcomplicating the model's architecture in the process. By systematically studying these modifications and their effects, it is anticipated to uncover valuable insights into the most effective ways of structuring the neural network model in this work to enhance its performance in cross-language clone detection.

The cosine annealing learning scheduler is a method for optimizing the learning rate of a cosine function, which is inspired by the concepts of simulated annealing and the form of the cosine function [17]. There is an idea behind this scheduler that aims to reduce the learning rate in a smoother, more gradual manner as training progresses. By adjusting the learning rate based on a cosine function in the range of $[0, /pi]$ over the course of an epoch, with each new one resetting the function, it is able to do this. Given a maximum learning rate lr_{max}, a minimum learning rate lr_{min}, the current epoch t, and the total number of epochs T, the learning rate at epoch t (lr_t) is calculated using the following equation:

$$\text{lr}_t = \text{lr}_{\text{min}} + 0.5(\text{lr}_{\text{max}} - \text{lr}_{\text{min}})(1 + \cos(\pi * t/T)) \tag{5.4}$$

It is important to note that the cosine annealing learning scheduler starts with a large learning rate and then gradually decreases it in accordance with the curve of the cosine function. As a result of this strategy, the model is able to explore the solution space very early in the training process, when the learning rate is high, and then refine the solution when the learning rate is reduced. An application of the cosine annealing learning scheduler for a Siamese network that uses InferCode embeddings for source code in Python and Java could help improve the performance of the network. During the training, the model can quickly explore different ar-

eas of the parameter space due to the high learning rate at the beginning of the training. With each passing epoch, the learning rate decreases gradually, allowing the model to fine-tune the parameters in areas where it has found promise over time, as the epochs progress. The advantage of this approach is that it might be able to increase the efficiency and performance of the model, enabling it to find better local minima within the loss landscape, thus possibly resulting in a more robust and accurate clone detection algorithm. The gradual reduction in learning rate ensures that the model does not overshoot the optimal solution while still allowing it to explore the solution space in a satisfactory manner without overshooting. As a result of this balance between exploration and exploitation, the performance of the model is often enhanced as a result. Below shows the result of adding cosine annealing activation function in our CLCD-I model. The accuracy and F1 score of the model did not change much with F1 score on test dataset is 0.60 and 0.70 accuracy.

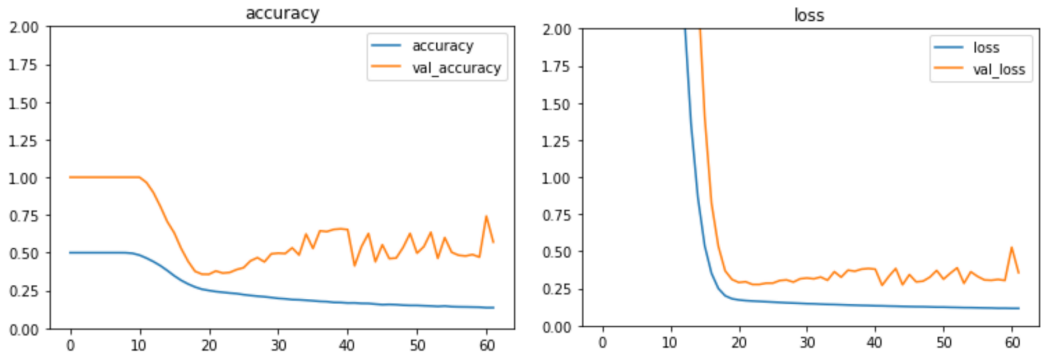


Figure 5.13: Siamese model with cosine annealing learning scheduler.

The performance stagnation issue observed when the cosine annealing layer was incorporated into our model. Initially, the vanishing gradient problem was pointed as the potential culprit behind this lack of improvement in the model's performance. There is a mechanism known as skip connections, also known as residual connections or shortcut connections,

which is used in neural network architectures to alleviate the problem of vanishing gradients in deep neural networks. By forming a direct path between the input and the output, these connections within a network's layer stack are able to backpropagate gradient information straight through the network, preserving the gradient information from earlier layers along the way.In a deep neural network, where the output of the l^{th} layer is denoted as $H_l(x)$, with x representing the input to the layer. In a conventional deep network, the output $H_l(x)$ is fed directly as the input to the succeeding layer. However, in a network incorporating skip connections, the input to the $(l+1)^{th}$ layer becomes a combination of $H_l(x)$ (the output from the l^{th} layer) and the original input x. This essentially allows the layer to learn from both the processed data from the previous layer as well as the original unprocessed input, thus enhancing its learning capability and mitigating the vanishing gradient problem. Below in Fig. 5.14 the skip connection layer was added to CLCD-I. In this work the original input 2 layers was passed ahead, so let the output of the l^{th} layer be denoted as $H_l(x)$, where x is the input to the l^{th} layer. In a network with skip connections that jump over two layers, the input to the $(l+2)^{th}$ layer is not just the output of the $(l+1)^{th}$ layer. Instead, it's the sum of the output of the $(l+1)^{th}$ layer, denoted as $H_{l+1}(x)$, and the original input x to the l^{th} layer. After careful observation, it was discerned that post the 20th iteration, the model began to exhibit a decrease in weight values. To counteract this phenomenon, residual connections was implemented specifically at layers which were exhibiting weight loss during the backpropagation process. The residual connections serve a dual purpose: not only do they help to preserve the weights during backpropagation, but they also facilitate the creation of multiple paths for backpropagation, thereby improving the capacity for weight updates. CLCD-I with skip connection reported 0.64 F1 score and 0.75 accuracy.

The experiments on CLCD-I were extended through the integration of a Convolutional Block Attention Module (CBAM) [77]. CBAM layer itself is developed from Squeeze and Excitation layer (EX) [35], which emphasizes important feature channels while reducing the influence of less relevant ones. The operational methodology of CBAM begins with

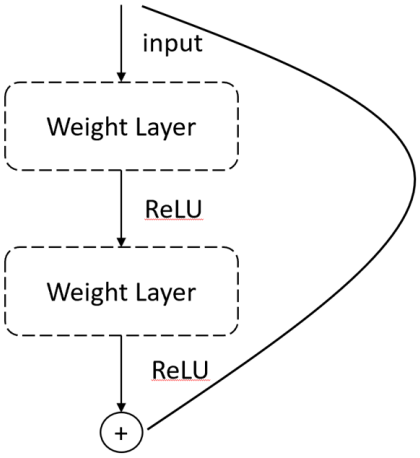


Figure 5.14: Skip connection passing input to 2 layers a head.

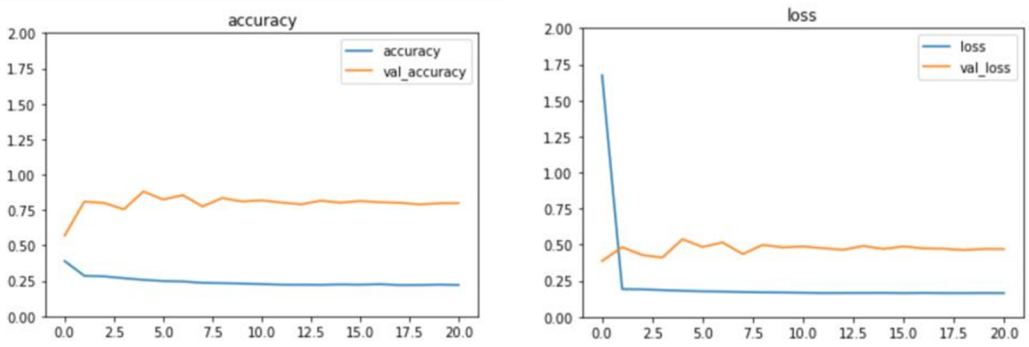


Figure 5.15: Siamese Architecture with skip connections.

receiving the input features, which are then relayed to a dedicated channel attention module, where the input features are then analyzed. In order to produce an intermediate result, the output of this module is multiplied with the feature input at the beginning of the process. Once this product is multiplied by the spatial attention output, it is sent to a spatial attention module, where it is once again multiplied by the spatial attention output in order to generate the final output. This approach offers an additional level of attention processing, both channel-wise and spatially, which enhances the ability of our model to recognize and select features. The CBAM layer in Fig. 5.16 was used. To explain what is going on inside CBAM layer, CBAM

takes the input feature, then pass it to a channel attention module and multiply it with input feature. Then the multiplied input feature with channel attention is passed to spatial attention model and again multiplied with output of spatial attention module.

CBAM channel attention module is composed of max pooling and average pooling layers. Max pooling operates by taking the maximum value from a certain window (or kernel) from the input. This operation essentially captures the most pronounced feature within the window, which is useful for recognizing patterns in the presence of transformations like a node in the AST of a source code is one position up or down, making it invariant to small transformations. In contrast, average pooling calculates the mean of all elements in the given window, essentially averaging out the features within the kernel. This leads to a reduction in abrupt changes while still preserving significant, widespread patterns in the feature set. This technique can be particularly useful for detecting broader, more general structures within the input, such as recurring patterns or widely distributed structures in the abstract syntax tree (AST) embeddings. The role of the MLP in the channel attention module of CBAM is to extract more complex and abstract representations of the source code from InferCode embeddings, thereby improving the effectiveness of cross-language code clone detection.

For example, it is assumed that a program code contains multiple instances of a for-loop iterating over an array. This recurring pattern creates a specific structure in the Abstract Syntax Tree (AST) of the code, visible across various subtrees. Each occurrence of the pattern might be slightly different, for example, due to variable names or different operations within the loop. When the code is processed and embedded through InferCode, these recurring patterns are reflected in the resulting vector representations. Applying average pooling on the InferCode embeddings, then the high-frequency changes, such as the differences in variable names or specific operations, are smoothed out. What remains is a more generalized representation of the recurring for-loop structure in the code, capturing the essential pattern rather than the specific details. This ability to reduce minor, abrupt changes and preserve wider, significant patterns allows the model to recognize and understand such recurring struc-

tures within different parts of the code, irrespective of their specific details. It can enhance the detection of broader patterns and structures within the input, leading to a more effective and robust model for cross-language code clone detection. To see the importance of max pooling, a source code snippet is assumed to involve a common programming structure such as a conditional statement (like an "if-else" structure). The "if-else" structure will have a specific representation in the Abstract Syntax Tree (AST) of the code, where specific nodes represent the conditional operation and its branches. When the code is processed and embedded through InferCode, this "if-else" structure becomes part of the resulting vector representations. Now, even if this "if-else" structure appears in different parts of the code (say once at the beginning, once in the middle, etc.), the max pooling operation can still recognize it as a significant feature. This is because max pooling identifies the most pronounced features within its kernel window, here capturing the common "if-else" pattern irrespective of its positional adjustments within the AST. This ability allows the model to recognize and understand such significant structures within different parts of the code, irrespective of their specific locations within the AST. It effectively provides an invariance to these minor positional transformations, which can significantly improve the robustness and effectiveness of the model for cross-language code clone detection.

For the subsequent phase involving the spatial attention module, the output from the channel attention module acts as input. Subsequently, the operations of max pooling and average pooling are applied to the incoming data. The resulting feature maps from both the MaxPooling and AveragePooling operations are subsequently merged via concatenation. Notably, the output channel derived from the concatenated results of both the MaxPooling and AveragePooling operations is a single channel. The spatial attention module focuses on the intra-relationship between different parts of the same feature dimension or "channel", effectively highlighting important parts within a feature dimension of the embeddings. The max pooling and average pooling are applied here within each feature dimension, generating two separate

"spatial" descriptors. These are then concatenated and used to reweight the importance of different parts within each feature dimension of the embeddings.

Despite the fact that both modules make use of max pooling and average pooling, they do so in different ways and with different purposes. Unlike the spatial attention module, which operates within each feature dimension to give a spatial descriptor, the channel attention module works across feature dimensions to give a channel-wise descriptor. Both of these components work together to provide a more sophisticated and comprehensive attention mechanism in processing InferCode embeddings of ASTs, improving the model's ability to identify code clones.

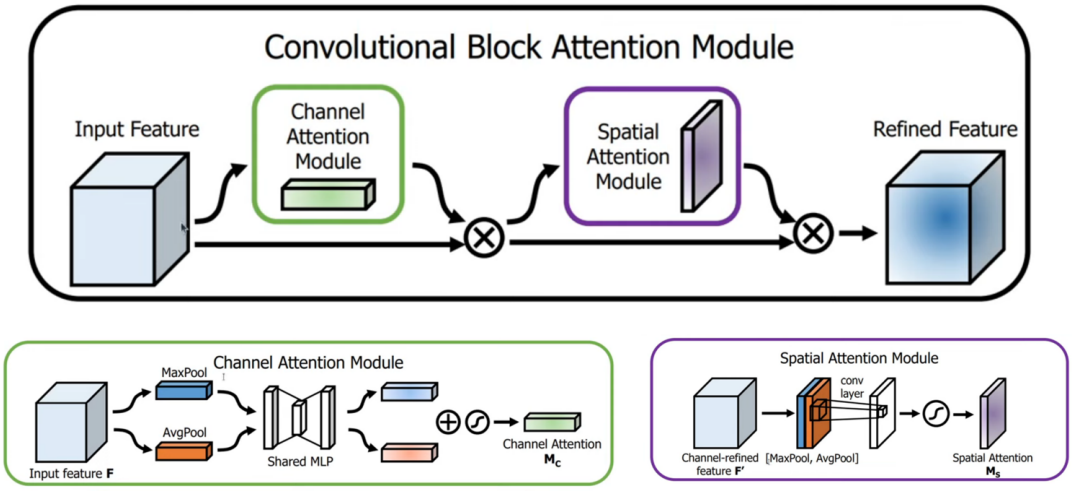


Figure 5.16: CBAM layer used in Siamese Architecture [77].

Below in Fig. 5.17 shows the performance of CLCD-I with CBAM layer. CBAM layer starts the CLCD-I by taking InferCode embeddings for both languages parallelly. The performance again did not change much and remained similar to our previous experiments with 0.67 accuracy and 0.80 F1 score.

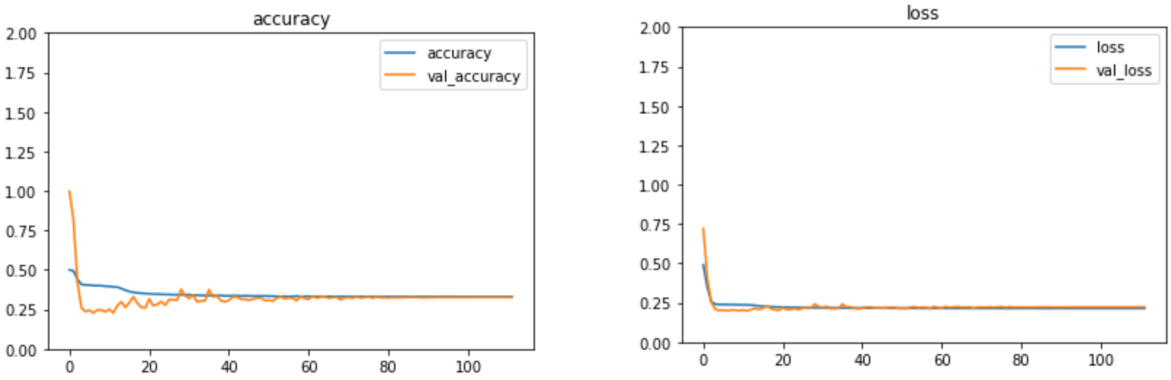


Figure 5.17: CBAM layer loss and accuracy used CLCD-I.

CHAPTER 6

CONCLUSION AND FUTURE WORK

CLCD-I is a new approach for detecting cross-language code clones using an InferCode-based source code embedding technique coupled with a Siamese neural network architecture and a quadratic contrastive loss function. The comparative performance of CLCD-I has been demonstrated through an experimental assessment, involving 20,000 pairs of similar code samples and an equal number of dissimilar code samples. The results demonstrate a significant improvement over the existing state-of-the-art methodologies by over 21%.

In the future, it is planned to extend CLCD-I's capabilities to be able to detect cross-language clones of Type IV in a wide range of programming languages. These clones, known as 'most elusive' clones, are characterized by similar logic, but differ considerably in their syntactic structure, which presents a unique challenge in detecting them. This task could be accomplished with the use of CFG embeddings, which present a promising approach.

In order to chart the future course, it is planned to focus on Type IV clone detection, as well as the addition of more languages to the system. For the purpose of enhancing the detection power for Type IV clones, we aim to leverage Control Flow Graph (CFG) embedding, which is a representation that is potentially more informative and nuanced as compared to the Abstract Syntax Tree (AST) used by InferCode for similar purposes. In order to detect Type IV clones, or programs that are similar in both semantics and syntax, it is crucial to capture the control flow of the program, as this will provide a semantic insight that could be crucial to detecting clones of Type IV programs. It is expected that by integrating embeddings from CFG with those of the AST, we will be able to construct a more comprehensive representation of the source code. As well as the above, attention mechanisms will be applied to distinguish which part of the embedding contributes the most to the clone as well as exploring the potential of applying SSL to the CFG is another exciting direction to explore.

CFG encapsulates control flow semantics, but it remains a challenge to embed them in a way that can be useful for SSL, in a way that is meaningful and generalized. Research in the future will be focused on developing innovative approaches to integrating SSL into CFG in the context of advancements in image and text domains, drawing inspiration from advancements in those domains. It is possible that such methods will lead to the development of a more refined model that will be able to detect intricate cross-language clones which have previously proved impossible to detect.

It should be noted that this work lays the foundation for further research into the application of self-supervised learning (SSL) to the detection of cross-language clones in the broader field of software engineering research. There is a good chance that SSL, which has demonstrated considerable success in image processing tasks, will prove to be a game-changer when it comes to source code analysis. The potential of harnessing SSL for inferring semantic similarities and structural similarities across code written in different programming languages could revolutionize the field of cross-language clone detection as continues to deepen our understanding of this domain.

[1] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. Compilers: Principles techniques and tools. 2007. *Google Scholar Google Scholar Digital Library Digital Library*, 2006.

[3] Marat Akhin and Vladimir Itsykson. Clone detection: Why, what and how? In *2010 6th Central and Eastern European Software Engineering Conference (CEE-SECR)*, pages 36–42. IEEE, 2010.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[7] Shaeela Ayesha, Muhammad Kashif Hanif, and Ramzan Talib. Overview and comparative study of dimensionality reduction techniques for high dimensional data. *Information Fusion*, 59:44–58, 2020.

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[9] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95. IEEE, 1995.

[10] Hamid Abdul Basit and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 513–516, 2007.

[11] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[12] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007.

[13] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 17–36. JMLR Workshop and Conference Proceedings, 2012.

[14] Riccardo Bonetto, Mattia Soldan, Alberto Lanaro, Simone Milani, and Michele Rossi. Seq2seq rnn based gait anomaly detection from smartphone acquired multimodal motion data. *arXiv preprint arXiv:1911.08608*, 2019.

[15] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems*, 6, 1993.

[16] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1186–1197. IEEE, 2021.

[17] Tristan Cazenave, Julien Sentuc, and Mathurin Videau. Cosine annealing, mixnet and swish activation for computer go. In *Advances in Computer Games*, pages 53–60. Springer, 2021.

[18] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[19] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[20] James R Cordy and Chanchal K Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220. IEEE, 2011.

[21] Jurafsky Daniel, Martin James H, et al. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. prentice hall, 2007.

[22] Arpita Das, Harish Yenala, Manoj Chinnakotla, and Manish Shrivastava. Together we stand: Siamese networks for similar question retrieval. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 378–387, 2016.

[23] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, pages 603–612, 2008.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[25] S Dhanabal and SJIJCA Chandramathi. A review of various k-nearest neighbor query processing techniques. *International Journal of Computer Applications*, 31(7):14–22, 2011.

[26] Angela Fan, Yacine Jernite, Ethan Perez, David Grangier, Jason Weston, and Michael Auli. Eli5: Long form question answering. *arXiv preprint arXiv:1907.09190*, 2019.

[27] Andrea Galassi, Marco Lippi, and Paolo Torroni. Attention in natural language processing. *IEEE transactions on neural networks and learning systems*, 32(10):4291–4308, 2020.

[28] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. Unsupervised representation learning by predicting image rotations. *arXiv preprint arXiv:1803.07728*, 2018.

[29] Robert Gold. Control flow graphs and code coverage. *International Journal of Applied Mathematics and Computer Science*, 20(4):739–749, 2010.

[30] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[31] Meng-Hao Guo, Tian-Xing Xu, Jiang-Jiang Liu, Zheng-Ning Liu, Peng-Tao Jiang, Tai-Jiang Mu, Song-Hai Zhang, Ralph R Martin, Ming-Ming Cheng, and Shi-Min Hu. Attention mechanisms in computer vision: A survey. *Computational Visual Media*, 8(3):331–368, 2022.

[32] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

[33] Vincent Josua Hellendoorn, Petros Maniatis, Rishabh Singh, Charles Sutton, and David Bieber. Global relational models of source code. 2020.

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[35] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[36] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability*, 70(1):304–318, 2020.

[37] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects. In *2012 19th Working Conference on Reverse Engineering*, pages 387–391. IEEE, 2012.

[38] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[39] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering*, 28(7):654–670, 2002.

[40] Cory J Kapser and Michael W Godfrey. Supporting the analysis of clones in software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):61–82, 2006.

[41] Satyendra Praneel Reddy Karri and B Santhosh Kumar. Deep learning techniques for implementation of chatbots. In *2020 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–5. IEEE, 2020.

[42] Shinji Kawaguchi, Takanobu Yamashina, Hidetake Uwano, Kyohei Fushida, Yasutaka Kamei, Masataka Nagura, and Hajimu Iida. Shinobi: A tool for automatic code clone detection in the ide. In *2009 16th Working Conference on Reverse Engineering*, pages 313–314. IEEE, 2009.

[43] Iman Keivanloo, Juergen Rilling, and Philippe Charland. Internet-scale real-time code clone search via multi-level indexing. In *2011 18th Working Conference on Reverse Engineering*, pages 23–27. IEEE, 2011.

[44] Yaser Keneshloo, Tian Shi, Naren Ramakrishnan, and Chandan K Reddy. Deep reinforcement learning for sequence-to-sequence models. *IEEE transactions on neural networks and learning systems*, 31(7):2469–2489, 2019.

[45] Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[46] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262. IEEE, 2006.

[47] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[48] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *14th working conference on reverse engineering (WCRE 2007)*, pages 170–178. IEEE, 2007.

[49] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[50] Yann LeCun and Fu Jie Huang. Loss functions for discriminative training of energy-based models. In *International workshop on artificial intelligence and statistics*, pages 206–213. PMLR, 2005.

[51] Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, and Dae-Kyoo Kim. Deep learning application on code clone detection: A review of current knowledge. *Journal of Systems and Software*, 184:111141, 2022.

[52] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[53] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. Modeling functional similarity in source code with graph-based siamese networks. *IEEE Transactions on Software Engineering*, 48(10):3771–3789, 2021.

[54] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics*, pages 87–94. IEEE, 2002.

[55] Lili Mou and Zhi Jin. *Tree-based convolutional neural networks: principles and applications*. Springer, 2018.

[56] Jonas Mueller and Aditya Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[57] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1026–1037. IEEE, 2019.

[58] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. Learning text similarity with siamese recurrent networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*, pages 148–157, 2016.

[59] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. Pmlr, 2013.

[60] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 518–528. IEEE, 2019.

[61] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[62] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[63] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[64] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, 2018.

[65] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[66] Julian Salazar, Davis Liang, Toan Q Nguyen, and Katrin Kirchhoff. Masked language model scoring. *arXiv preprint arXiv:1910.14659*, 2019.

[67] Michael Lee Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000.

[68] RW Sebesta. Concepts of programming languages, 10th, international ed, 2012.

[69] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[70] Chawin Sitawarin and David Wagner. On the robustness of deep k-nearest neighbors. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 1–7. IEEE, 2019.

[71] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal processing magazine*, 25(2):128–131, 2008.

[72] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE, 2014.

[73] Md Sharif Uddin. *Dealing with clones in software: A practical approach from detection towards management*. PhD thesis, University of Saskatchewan, 2014.

[74] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[75] Wenxuan Wang, Wenxiang Jiao, Yongchang Hao, Xing Wang, Shuming Shi, Zhaopeng Tu, and Michael Lyu. Understanding and improving sequence-to-sequence pretraining for neural machine translation. *arXiv preprint arXiv:2203.08442*, 2022.

[76] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *2011 18th Working Conference on Reverse Engineering*, pages 35–44. IEEE, 2011.

[77] S Woo, J Park, JY Lee, and I So Kweon. Cbam: convolutional block attention module. in proceedings of the european conference on computer vision (eccv): 3-19, 2018.

[78] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pages 10799–10808. PMLR, 2020.

[79] Yuan Yuan, Weiqiang Kong, Gang Hou, Yan Hu, Masahiko Watanabe, and Akira Fukuda. From local to global semantic clone detection. In *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, pages 13–24. IEEE, 2020.

[80] Jie Zeng, Kerong Ben, Xiaowei Li, and Xian Zhang. Fast code clone detection based on weighted recursive autoencoders. *IEEE Access*, 7:125062–125078, 2019.

[81] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[82] Yong Zhang, Dan Li, Yuheng Wang, Yang Fang, and Weidong Xiao. Abstract text summarization with a convolutional seq2seq model. *Applied Sciences*, 9(8):1665, 2019.

[83] Arseny Zorin and Vladimir Itsykson. Recurrent neural network for code clone detection. *Software Engineering and Infomation Management*, 47, 2018.