Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

A study of machine learning approaches to cross-language code clone detection

言語間でのコードクローン検出のための
機械学習的な手法についての研究

Daniel PEREZ HERNANDEZ
ペレズヘルナンデズ ダニエル

Supervisor: Professor Shigeru Chiba

January 2018

Abstract

While clone detection across programs written in the same programming language has been studied extensively in the literature, the task of detecting clones across multiple programming languages is not covered as well, and approaches based on comparison cannot be directly applied. In this thesis, we present a clone detection method based on supervised machine learning able to detect clone across programming languages. Our method uses an unsupervised learning approach to learn token-level vector representations and an LSTM-based neural network to predict if two code fragments are clones. To train our network, we present a cross-language code clone dataset — which is to the best of our knowledge the first of its kind — containing more than 50000 code fragments written in Python and Java. We show that our method can detect code clones between Python and Java with high accuracy. We also compare our method to state-of-the-art tools in single-language clone detection and show that we achieve similar accuracy.

概要

同一の言語でのクローン検出はよく研究されているものの、複数の言語間でのクローン検出を行う研究は少ない。またトークンや構文木の比較を用いた手法が直接利用できない。本研究では、教師あり機械学習を用いて言語間でのクローンを検出する方法を提案する。本手法では、教師なし学習を用いてトークンのベクトル表現を計算した上で、LSTMベースのニューラルネットワークで２つのプログラムがクローンであるかどうかを判定する。ネットワークを学習させるために、５万以上のプログラムを含む複数の言語間でのコードクローンのデータセットを作成した。JavaとPython間のコードクローンを高い精度で検出できることを示す。
また、同一言語間でのクローン検出の精度を既存のツールと比較する。

Contents

Chapter 1

Introduction

Code clones are fragments of code, in a single or multiple programs, which are similar to each other. Code clones can emerge for various reasons, such as copy-pasting or implementing the same functionality as another developer. Code clones can decrease the maintainability of a program, as it becomes necessary to fix an error in all the places where the code was copied. However, detecting these code clones — which is often referred to as the task of code clone detection — is a difficult task and has been extensively researched in the literature. Some systems focus on finding clones inside a single project, while other task try to detect clones in larger ecosystem, by for example trying to detect clones in open-source development websites such as GitHub. With the variety of tools and approaches proposed for clone detection, the task of code clone detection has also found various applications, such as helping for code search, malware detection, copyright infringement detection, and many others. However, although there is a very large amount of tools that have been developed for the task of clone detection, most of these have been developed to detect clones in programs written in the same programming language, and the task of detecting code clones for programs written in different languages has not been studied as well in the literature.

In recent years, the number of programming languages typically used by a developer has increased [1], and developers tend to switch between programming languages much more often than a decade ago. There are many possible explanations to this phenomenon. One reason is that many companies use different languages for prototyping and production, and programmers therefore tend to use a scripting language such as Python to develop a prototype and a faster language such as C++ for the production code. Another explanation is that native applications often need to be written in different programming languages: iOS applications can be written in Swift or Objective C, Android applications require a language running on the JVM, such as Java or Kotlin, and application running in the browser need to be written in JavaScript. Yet another potential reason for this proliferation of languages is the democratization of micro services. As multiple parts of a single system often run in total isolation from each other in their own process or container, the need to write everything in a single language is not as high as it used to be with monolithic applications and teams therefore tend to choose the programming language the most adapted for a given task, and having a system with a websocket server running JavaScript code and another server performing machine learning tasks using Python is not rare.

Given such a context, where it is common for a system to be implemented using different programming languages, code clones between part of the systems implemented in different programming languages will eventually emerge, causing the same kind of maintainability issues as code clones in a program written in the same programming language. To help improving this situation, the question of whether we can detect code clones in programs written in different programming languages is therefore important.

In this thesis, our goal is to find a method to detect code clones written in different programming languages. Given the variety of languages and the potential difference between them, rule-based comparison approaches seem challenging to apply in this context. We therefore decided to focus on machine learning based techniques to perform this task. We design a system to perform this task, collect data to train the system, and evaluate our system to see how well it performs. In this thesis, we present two main contributions.

- We present a system based on supervised machine learning capable of detecting clones across programming languages. We train and evaluate our model using programs written in Python and Java, and obtain promising results in terms of precision and recall. Our system being language agnostic, the only requirement for being able to support new programming languages is to be able to collect code clones data written in the desired language.

- We present a cross-language code clones dataset containing about 50000 files written in Java and Python. Although code clones dataset exist for programming languages such as Java, this is to the best of our knowledge the first dataset with cross-language code clones. We collected the data for this dataset by scraping competitive programming websites and used the data provided by the websites to extract code clones.

We also present in detail how we transform the programs to be able to feed them to a machine learning model and show how applying natural language processing methods can improve our results.

The rest of this thesis is organized as follow. In Chapter 2, we give some background about the task we are trying to solve by presenting the necessary concepts, giving motivating examples and discussing the related work in this area. In Chapter 3, we give details about our proposal, we describe the different parts of our system and explain how our machine learning model works and how we train it. In Chapter 4, we describe the different experiments we performed, show and discuss the results we obtained. Finally, in Chapter 5, we summarize what we have presented in this thesis and give some directions for future work in this area.

Chapter 2

Background

2.1 Source code representation

The way we represent source code is very important in code clone detection, as it affects greatly the algorithms that may be used to detect clones.k In this section, we will explain the different ways in which source codes can be represented.

2.1.1 String representation

The most common representation for a program is its string representation. It is the representation that developer actually uses when editing a program, and is therefore the easiest to obtain, as it is simply the raw program. Listing 2.1 is a function taking two numbers as input and returning their sum.

```
1 int add(int a, int b) {
2   return a + b;
3 }
```

Listing 2.1. `add` function

The string representation of the above program would simply be

```
1 "int␣add(int␣a,␣int␣b)␣{\nreturn␣a␣+␣b;\n}"
```

2.1.2 Token representation

While the string representation is an array of characters, the token representation is an array of strings where each string in the array is a token of the program. It is typically the representation that a lexer would return while processing the source program. The above program would have the following token representation.

```
1 ["int", "add", "(", "int", "a", ",", "int", "b",
2  ")", "{", "return", "a", "+", "b", ";", "}"]
```

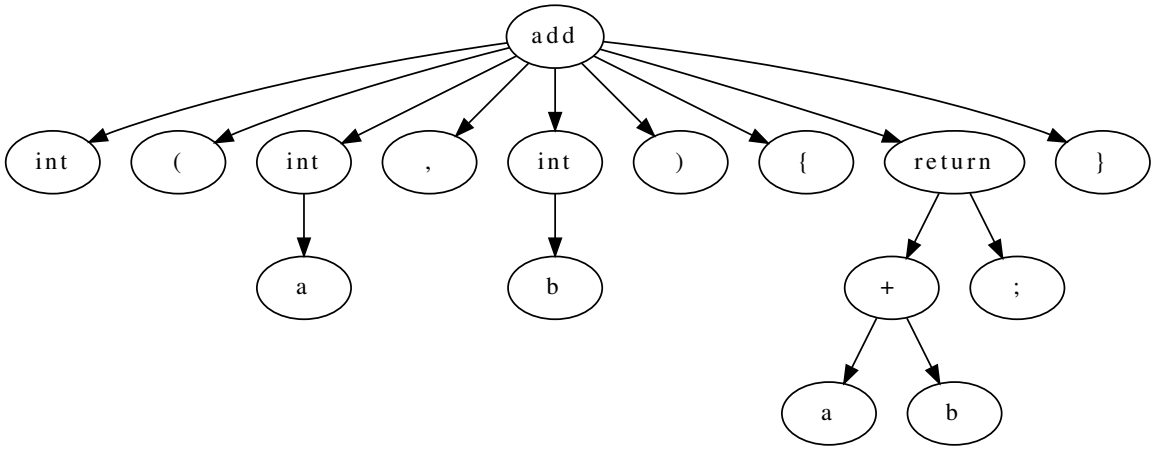

Figure 2.1. Concrete Syntax Tree of `add` function

Token representation may, or may not keep whitespaces and other formatting characters, but except for programming languages where whitespaces are significant, such as Python, they are usually not included in the output.

2.1.3 Tree representation

There are two main tree representations for programs: concrete syntax trees and abstract syntax trees. We will describe the two different type of trees and there differences.

Concrete Syntax Trees

Concrete syntax trees are concrete in the sense that they keep all the syntactic tokens of the program, such as commas, semi-colons or parentheses. It therefore contains exactly the same tokens as the token representation, but the tokens are organized as a tree, where the root is usually the entry of the program, the nodes are usually statements or expressions and the leaves are identifiers or literals. Figure 2.1 shows how Listing 2.1 would be represented as a concrete syntax tree.

Abstract Syntax Trees

Abstract Syntax Trees (AST) are also a tree representation of the source code, but as opposed to the concrete syntax tree, they do not preserve the host language syntax, such as parentheses, and instead express these using the structure of the tree. AST are typically the what a parser returns when processing the source code, and are one of the most common representation for various tasks on programs, including clone detection. Figure 2.2 shows the AST for Listing 2.1.

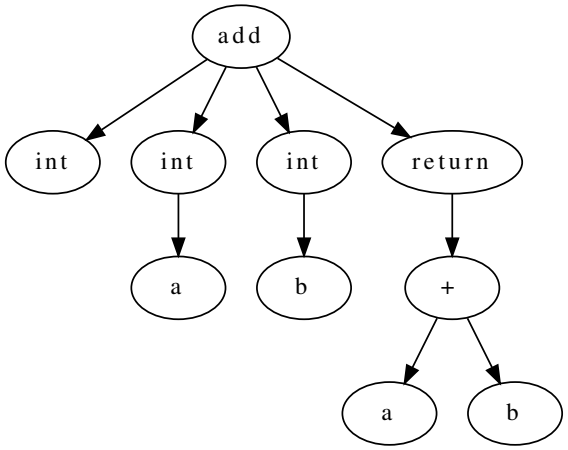

Figure 2.2. Abstract Syntax Tree of `add` function

2.2 Code clone fundamentals

In this section, we will first describe the different types of code clones that appear in the literature, and that we will be using in this thesis. We will then view the different approaches to code clone detection that can be found in the literature.

2.2.1 Types of code clones

It is important to define what is implied when two code fragments are defined as clones. In the literature, code clones are usually classified into four different types [2], from Type I to Type IV, where Type I is the "strongest", meaning here that the two code fragments are the most similar, while Type IV is the "weakest" — the two code fragments share less in common. We will now give a more precise definition of each code clone type.

Type I Clones

Type I code clones types are clones where the code fragment are the same, except for some variations that do not affect the logic, such as differences in whitespaces or comments. In the literature, Type I clones are sometimes also referred to as exact clones. For example, let us consider the following code fragment.

```
1 int fibo(int n) {
2   if (n <= 1) {
3     return n;
4   }
5   return fibo(n - 1) + fibo(n - 2);
```

```
6 }
```

Listing 2.2. Fibonacci function

The following code fragment is considered to be a Type I clone.

```
1 int fibo(int n) {
2   if (n <= 1) { return n; } // base case of recursion
3   return fibo(n-1)+
4          fibo(n-2);
5 }
```

Although, as with the above example, line-by-line comparisons may not always yield correct results for Type I clones, it is worth noting than if the two fragments are pre-processed with a lexer, a simple token comparison should normally be enough to detect this type of clones.

Type II Clones

Type II clones include the changes that can be involved in Type I clones, such as layout and comments, but also include other changes, such as changes in user-defined identifiers — variable or function names. For example, the following snippet would be a Type II clone for Listing 2.2:

```
1 int fib(int x) {
2   if (x <= 1) { return x; }
3   return fib(x - 1) + fib(x - 2);
4 }
```

The function and parameter names, as well as the layout are different, but the logic is exactly the same.

Type III Clones

Type III clones can have all the changes from type I and type II clones, but may also have statement insertions, deletions or replacements. Although Type III clones are usually copied from the same source code, there is no agreement on a threshold on the number of changes to consider or not two code fragments as type III clones. Some researchers therefore classify type III code clones in categories such as very strong type III or weakly type III [3]. Here is an example type III clone for Listing 2.2.

```
1 int fib(int x) {
2   int a = x;
3   if (a <= 1) {
4     return a;
5   }
6   return fib(a - 1) + fib(a - 2);
7 }
```

In the above code fragment, the code is completely copied from Listing 2.2, except for a change in the identifier names, but the second line has been inserted, making this code fragment a type III clone.

Type IV Clones

Type IV clones, also referred to as functional code clones, are two code fragments implementing the same functionality with different approaches. Unlike type I to III clones, type IV clones are usually not a result of code copying, but rather of two programmers implementing the same functionality. For example, as opposed to the recursive implementation of Fibonacci in Listing 2.2, the following iterative implementation could be defined to be a type IV clone.

```
1 int fib(int n) {
2   int a = 0, b = 1;
3   for (int i = 0; i < n; i++) {
4     int tmp = a + b;
5     a = b;
6     b = tmp;
7   }
8   return a;
9 }
```

Listing 2.3. Iterative Fibonacci

The two code fragments will return the same result for any $n \geq 0$ and are therefore functional clones. Although the implemented functionality is the same, there are no other common point in the implementations. Furthermore, Listing 2.2 and Listing 2.3 clearly have different time complexities, but to the best of our knowledge, there are no established convention to whether this kind of complexity changes should be taken into account when deciding if two code fragments are, or not, type IV code clones. In this thesis, we will therefore assume that as long as two code fragments implement same functionality, they are type IV clones.

2.2.2 Approaches to clone detection

There exist various methods to clone detection, which take advantage of the different source code representations described in section 2.1. We will present the major approaches found in the literature.

Text-based detection

Text-based clone detection are based on the string representation of the program source code described in 2.1.1. While some text-based techniques use the source code almost as is and perform either line-by-line or global string comparisons to find clones, some techniques do apply some preprocessing steps [4], such as removing white spaces, before actually passing the source code to a string matching algorithm.

Token-based detection

Token-based detection techniques use the token representation of the program described in 2.1.2. Token-based approach are usually more robust to formatting and layout changes compared to text-based approaches. Some token-based clone detection tools, such as CCFinder [5] first normalize the tokens and replace all application specific tokens such as user-defined identifiers. On the other hand, other tools such as SourcererCC [3] keep this information and use it to detect if multiple code fragments are clones or not.

Tree-based detection

Tree-based techniques make use of the AST representation of the program described in 2.1.3. These techniques first use a parser to generate the AST of a program, then the ASTs are searched for similar sub-trees and similar sub-trees are marked as clones. Searching for similar sub-trees using normal tree comparisons algorithms takes at least $\mathcal{O}\left(n^3\right)$ [6], where n is the number of nodes. Industrial software can have millions of line of codes, and therefore n could be in the order of 10 million. To avoid this issue, many tree-based clone detection techniques [6, 7] use hash functions on sub-tress to find clones more efficiently. The idea is to assign each sub-tree a hash-value, and use this value, usually with bucketing techniques, to quickly search for code clones.

2.3 Machine learning fundamentals

In this section, we will present the basics of machine learning that will be needed in order to understand our approach. We will first give a general overview of the learning process of neural networks, then we will describe the two models we are using as a base to our work in this thesis. The first model will present is the skipgram model, which is typically used in natural language processing (NLP) to assign a vector representation to a word. Next, we will give an overview of recurrent neural networks (RNN), and in particular long-short term memory (LSTM) networks, which are extensively used in NLP to assign a vector representation to a sentence.

2.3.1 Neural networks

In this subsection, we will show how are neural network composed, and what is the normal process to train them. Although most of what we explain here apply to most machine learning algorithms, we will focus on feed-forward neural networks.

Computations of a neural network

A neural network [8] is usually composed of an input layer, which is where the input data is fed, one or more hidden layers, and an output layer, which represents the value outputted by the neural network. Each layer in the network is composed by units, also sometimes referred to as neurons, which are the computing elements of the network. Each unit is usually composed of two different parts: a first function f, which reduces a vector into a single scalar value, and a function g, called "activation function" which transforms the result of f and produces the unit

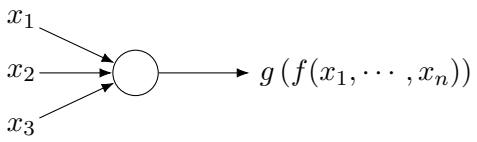

Figure 2.3. Unit of a neural network

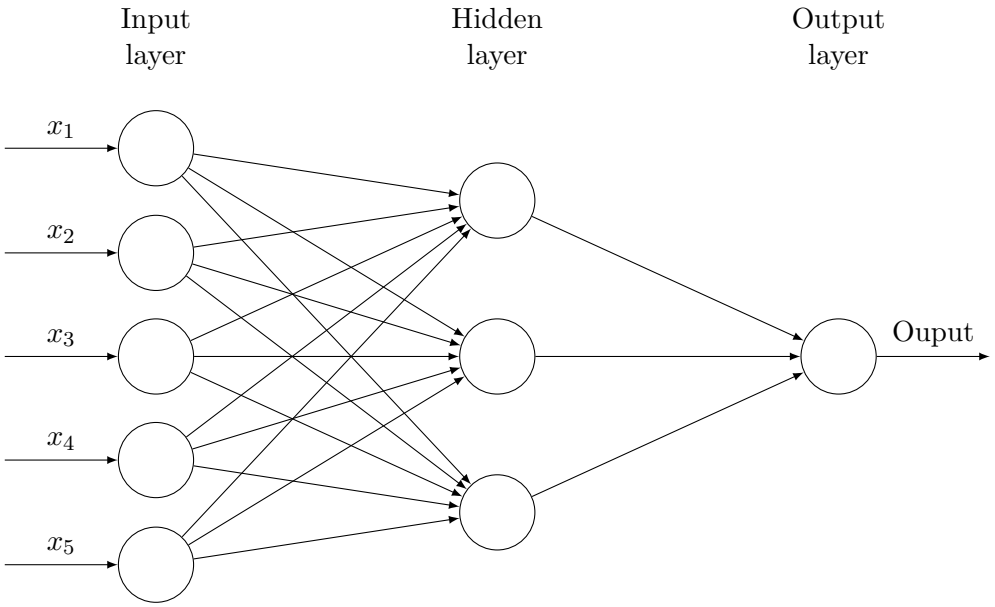

Figure 2.4. Neural network

output. This function is generally non-linear, otherwise the model ends up to be composed of only linear function and becomes as well a linear function. A unit of a neural network is illustrated in figure 2.3. Each layer is connected to the next one by using the output of each unit of the previous layer as an input for each unit of the next layer. This connection is illustrated in figure 2.4.

The function f is a matrix multiplication where all the weights of this matrix are parameters of the neural network model. Matrix weights are shared by all the units in a layer. In figure 2.4, given $W^{(1)} \in \mathbb{R}^{5\times 3}$ is the matrix weights of the hidden layer, W_{mn} is the value at row m and column n in this matrix, g_1 is the activation function of the hidden layer, and a_1, a_2 and a_3 the output of each unit in the layer, we will have the following.

$$
\begin{aligned}
a_1 &= g\left(W^{(1)}_{11}x_1 + \cdots + W^{(1)}_{51}x_5\right) \\
a_2 &= g\left(W^{(1)}_{12}x_1 + \cdots + W^{(1)}_{52}x_5\right) \\
a_3 &= g\left(W^{(1)}_{13}x_1 + \cdots + W^{(1)}_{53}x_5\right)
\end{aligned}
$$

In practice, given $x = (x_1, \cdots, x_5)$ and $a = (a_1, a_2, a_3)$, the above is usually computed as follow,

with a single matrix multiplication and a vectorized implementation of the function g.

$$a = g\left(xW^{(1)}\right)$$

Finally, given $W^{(2)} \in \mathbb{R}^{3\times 1}$ the weights of the output layer and g_2 the activation of the output layer, the output $\hat{y}$ of the neural network will be computed as follow.

$$\hat{y} = g_2\left(g_1\left(xW^{(1)}\right)W^{(2)}\right) \tag{2.1}$$

Training of a neural network

Training a neural network is searching for the optimal weights to approximate the function the neural network should learn. In the above example, this process consists in searching values for $W^{(1)}$ and $W^{(2)}$. The "optimal" weights of a neural network are defined with respect to a loss function representing how far the neural network output is from the expected correct answer, which is chosen depending on the task. Some common loss functions include mean squared error (MSE) 2.2 or cross-entropy error (CEE) 2.3. Given a set of outputs $\hat{Y} = \{\hat{y_1}, \cdots, \hat{y_n}\}$ and their respective expected values $Y = \{y_1, \cdots, y_n\}$, the errors are defined as follow.

$$MSE(\hat{Y}, Y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.2}$$

$$CEE(\hat{Y}, Y) = \frac{1}{n}\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i) \tag{2.3}$$

Mean squared error is typically used when there is an order relation between the possible outputs while cross-entropy error is mostly used for classification tasks where the output can either be correct or incorrect.

The most common way to update the weights of the neural network is called the back propagation algorithm, and consists in computing the derivatives of the loss function with respect to the weights of the model using the derivative chain rule, and to use these to update the weights with optimization methods such as the gradient descent. For example, given a loss function $L(W)$, to update $W^{(2)}$ in the above network, we would compute the error derivative $\partial L(W)/\partial W^{(2)}$, multiply the result with a learning rate α, which is a hyper parameter of the model, and subtract the result from the current value of $W^{(2)}$. This process is usually repeated until the error converges.

Evaluating a model

When training a neural network, or any kind of machine learning algorithm, we need a way to know if the model produces the results we expect or not — a way to evaluate the quality of the model. The error, also called loss, we presented above can be an indicator of how well a performing but it is unfortunately not easy to interpret as depending on the used function the range can vary largely and it is difficult to significantly compare the values produced by different errors functions. We therefore tend to use measures other than the loss to evaluate the quality

of the model. We will present different measures that are commonly used. We will only present measures used for binary classification here, but all of these measures can be generalized in some way to also work for multi-class classification.

To define the different measure we will use, we will first introduce the notions of true positive, false positive, true negative and false negative.

True positive A sample predicted as true and was actually true

False positive A sample predicted as true but was actually false

True negative A sample predicted as false and was actually false

False negative A sample predicted as false but was actually true

We will assume we have a set of predicted samples S with $|S| > 0$, and we will note the set of true positives as T_p, false positives as F_p, true negatives as T_n and false negatives as F_n.

One of the most common and simplest measure to understand is the accuracy of a model. The accuracy is, as shown in equation 2.4 the number of samples correctly predicted divided by the total number of samples predicted.

$$\text{accuracy} = \frac{|T_p| + |T_n|}{|S|} \tag{2.4}$$

The accuracy has a range of $[0, 1]$, and a higher accuracy is better. It is widely used to evaluate a model, but may have shortcomings depending on the dataset. A common issue with the accuracy measure is with unbalanced datasets — a dataset with much more negative samples than positive samples, or vice-versa. For example, given a dataset with 99% of negative samples, if a model returns a negative result for any sample, it will achieve 99% accuracy but will not be useful at all.

To be able to get a better understanding of a model performance, especially when the dataset is unbalanced, the precision and recall measures are often used. The precision of a model, given by equation 2.5 is the ratio of samples correctly predicted to the total number of samples that have been evaluated as positive.

$$\text{precision} = P = \frac{|T_p|}{|T_p| + |F_p|} \tag{2.5}$$

The precision also has a range of $[0, 1]$, with 1 being the best achievable precision, meaning that all samples predicted as positives were actually positives. However, the precision by itself may not be enough: if a model predicts a single sample as positive, and the sample actually turns out to be positive, the precision will be 1 but the model might be missing a lot of other positive samples. This is why the recall measure is also often used. The recall measure, given by equation 2.6, is the ratio of correctly predicted samples to the total number of positive samples.

$$\text{recall} = R = \frac{|T_p|}{|T_p| + |F_n|} \tag{2.6}$$

The precision also has a range of $[0, 1]$, and 1 is also being the best achievable recall. If a model predicts all samples to false, as in the example we gave for the accuracy, the recall will therefore be 0, as obviously there will be no sample in the true positives set. However, the recall by itself is not enough, as predicting all samples to be positive would be enough to achieve a recall of 1. The recall and precision are therefore usually used together. To avoid having to look at two different measures when evaluating a model, the F1-score — which is the harmonic mean of the recall and the precision as shown in equation 2.7 — is often used.

$$\text{F1-score} = \frac{2}{\frac{1}{P} + \frac{1}{R}} = 2\frac{P \cdot R}{P + R} \tag{2.7}$$

As the F1-score is the harmonic mean of two measures with a range of $[0, 1]$, it also has the same range, and as for the precision and recall, a higher F1-score means that the model is performing better. The main reason for using a harmonic mean instead of arithmetic mean is that it allows to penalize the score if one of the value is much lower than the other, therefore forcing both the recall and the precision to be high in order to have a high score.

2.3.2 Skipgram model

As mentioned above, skipgram is used to assign a vector in $\mathbb{R}^d$, known as a word embedding, to a word, where d is the number of dimensions wanted for each word vector. This algorithm has first been presented in 2013 by Mikolov & al. [9] and is now used in many natural language processing tasks as one of the word2vec algorithms.

To understand the motivations behind this algorithm, and other algorithms used to learn word embeddings, we must first understand how words used to be represented in natural language processing tasks. The most common approach was to use a finite vocabulary of size $|V|$, and to assign an identifier to each word in this vocabulary. For example, the word "network" could have the id 123, and would be represented as a vector in $\{0, 1\}^{|V|}$ where only index 123 of the vector would be 1 and all the other indexes would be 0. This approach has two disadvantages: each word is represented by a very large and sparse vector, and the representation of each word do not have any semantic properties, so "network" and "networks" vector representations would not be any closer than "network" and any other word representation.

The motivation behind skipgram is to address this issue by encoding each word as a vector in $\mathbb{R}^d$ where d is not dependent of $|V|$, in such a way that each word embedding has some semantic so that, for example, similar words have a smaller distance in $\mathbb{R}^d$ than unrelated words.

We will now describe how the skipgram algorithm actually computes the vector embeddings for each word. The algorithm works in an unsupervised fashion and usually uses a large amount of text written in the language of the vocabulary. For each sentence, a set of training samples composed of a "context" word and a "target" word. The context word is a word in the sentence, while the target word is a word around the context word, which is distant by at most a predefined window size. For example, if the window size is 2, and we have the following sentence,

the quick brown fox jumped over the lazy dog

the target words for the context word "fox" will be "quick", "brown", "jumped" and "over". These samples are used to train a neural network with a single hidden layer. The word embeddings

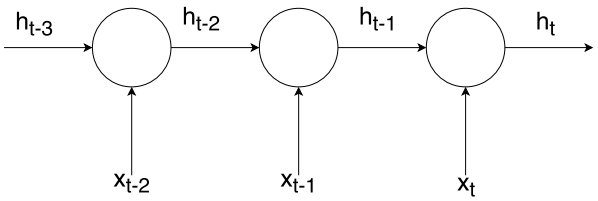

Figure 2.5. Unrolled Recurrent Neural Network

are the hidden layer of the network and have dimension $\mathbb{R}^{|V|\times d}$. The network is trained by using the context word index to select a vector from the hidden layer weight and using a cross-entropy error function so that when the word is a target, the label should be 1 and 0 otherwise. Also this could be done by using a softmax output, this would require to compute softmax on all the vocabulary, which can be extremely large. To reduce the computation cost, a common approach is to used negative sampling. The negative sampling approach only computes the output for the target word, as well as a predetermined number of noise word — word not in the window of the context — and uses the sum of these errors to train the model.

2.3.3 Recurrent Neural Networks

While word embeddings work well to represent a single word, and to model the similarities between words, it is not enough to encode a whole sentence. There are many approaches to encode a sentence into a vector, and Recurrent Neural Networks (RNN) is one of them. The main idea of RNNs is to reuse the previous output as an input, as shown in 2.5. Typically, each x_t is a word in the sentence, encoded either as a one-hot vector or as a word embedding. In its simplest form, a RNN will compute its output using equation 2.8, where W, U and b are parameters of the model. The parameters are shared across all time steps.

$$h_t = \tanh\left(Wx_t + Uh_{t-1} + b\right) \tag{2.8}$$

Although this models seems promising, it suffers a major issue — it is very difficult to capture long-term dependencies because of the vanishing gradients issue [10]. When computing the gradients to back-propagate the error through the network, the gradient for ∂x_t with respect to ∂x_k, used in the chain rule, can be expanded as follow.

$$\frac{\partial x_t}{\partial x_k} = \prod_{t \geq i > k} \frac{\partial x_i}{\partial x_{i-1}} \tag{2.9}$$

The term $\partial x_i / \partial x_{i-1}$ in equation 2.9 is less then 1 [10], which means that the value will converge to 0, and error will not propagate all the way through the network, hence the vanishing gradient issue.

To overcome this issue, Long Short-Term Memory (LSTM) networks [11] are often used instead of vanilla RNN described above. Instead of equation 2.8, LSTMs use equations 2.10

to 2.15 to compute the output value. All W, U and b in the equations are parameters of the model.

$$f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}\right) \tag{2.10}$$

$$i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)}\right) \tag{2.11}$$

$$u_t = \tanh\left(W^{(u)}x_t + U^{(u)}h_{t-1} + b^{(u)}\right) \tag{2.12}$$

$$o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)}\right) \tag{2.13}$$

$$c_t = i_t \odot u_t + f_t \odot c_{t-1} \tag{2.14}$$

$$h_t = o_t \odot \tanh(c_t) \tag{2.15}$$

In the above equations, σ is the sigmoid function defined by equation 2.16,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.16}$$

and the different values have the following meaning.

f_t — forget gate — how much of the previous state should be used

i_t — input gate — how much of the input should be used

u_t — update gate — the value of the current input

o_t — output gate — how much of the current result should be outputted

c_t — memory cell — the state of the network

h_t — hidden layer — the output of the LSTM cell

Given $g^n(x_t, h_{t-1}) = W^{(n)}x_t + U^{(n)}h_{t-1} + b^{(n)}$, figure 2.6 shows how the inputs flow in equations 2.10 to 2.15 to generate the outputs of an LSTM cell. In an LSTM network, the state is not passed through h_t as with a plain RNN, but through c_t, called the memory cell. Due to the additive nature of c_t, the error will also be propagated in an additive manner when computing the derivative for c_t [11], avoiding the vanishing gradient problem encountered when using plain RNNs.

2.4 Motivating example

With the proliferation of programming languages, a single project or application is often written using multiple programming languages. For example, a project having a native client for the web, Android and iOS will most likely have at least a part of it written in JavaScript, a part written in Java or Kotlin and another part written in Swift or Objective C. A common issue with such a project is having a large amount of logic duplicated for each client. In many cases, business logic for an application with multiple frontends can be extracted to the backend to avoid duplication.

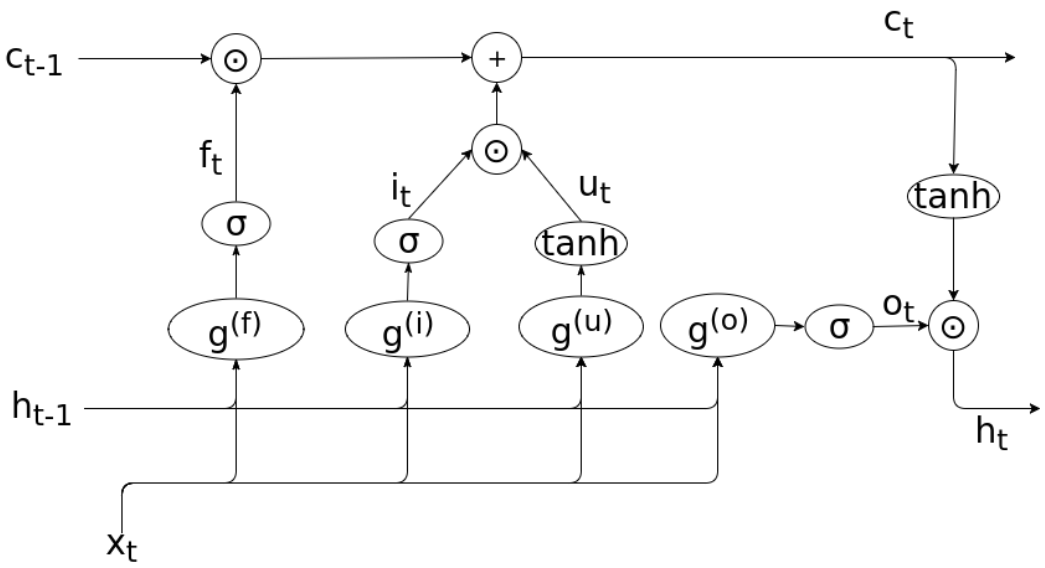

Figure 2.6. LSTM cell

This issue is difficult to tackle from both an organizational point of view as well as a technical point of view. From the organizational point of view, a common project development structure would have a project manager making high level and mostly non-technical decisions, while a team would be assigned for each component of the project, for example a team would work on the backend API, a team on the Android application, another team on the iOS application and finally a team on the web client. Although thorough discussion between the team can help to mitigate code duplication on the client side up to a certain point, it remains a very tedious issue to track.

As an example, we will present a simplified version of a case we actually encountered while building a web service. The frontend user interface, shown in figure 2.7, consists of a page displaying recent messages grouped by sender.

As the conversation grouping feature was not part of the initial application, and has been added later on, the API would simply have an HTTP endpoint `GET /messages` returning a list of the user messages. Given this restriction on the API side, the frontend clients need to group the messages by conversation, leading to duplicated code show in listings 2.4, 2.6 and 2.5. All of these listings implement the following piece of functionality

1. Sort messages from the oldest to the newest
2. Group messages by the ID of the sender
3. Sort each group from the oldest to the newest based on its newest message
4. Return the result as a collection of collections of messages

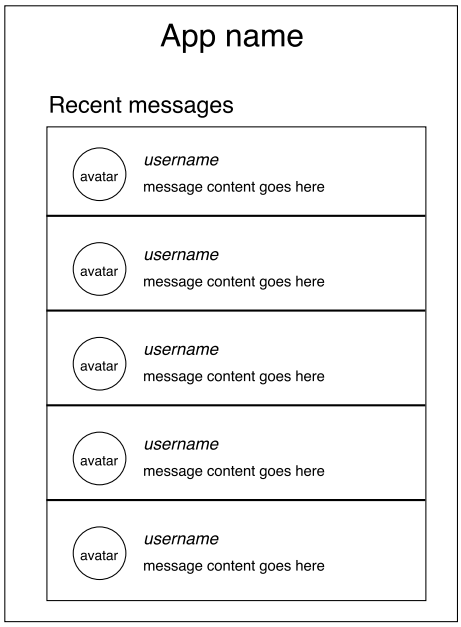

Figure 2.7. Study case application user interface

Listing 2.4 is implemented in JavaScript, listing 2.5 is implemented in Java and listing 2.4 is implemented in Swift which are respectively used for the web client, the Android application and the iOS application. Although the logic is implemented in exactly the same way, each language have its particularities and there are therefore some differences in the implementation of each snippet. It is also worth noting that there are also semantic differences in the different implementations that can be more tedious to notice at first look. The first step in all the implementations is to sort the list of messages received as an argument. All three implementations use some `sort` implementation available in the standard library of each language, but this creates an important semantic difference — Swift implementation does not modify the argument, while Java and JavaScript implementation modifies it. This side effect can be particularly difficult to notice in the JavaScript implementation as the return value of the `sort` method is used, and it is not clear that the function argument is actually modified without having to search the documentation. The next step of the implementation is to group all the messages by their `senderId`. This makes use of a `map`-like data structure, which is available as a builtin of all three languages. All snippets loop over all the messages, maps the `senderId` key to an empty list if it does not already exist, and adds the current message to the `senderId` list. All the grouped messages are then collected in a list of list, using the map `values` function when available, or looping through the map otherwise. Finally, the list of list is sorted using the `sentAt` value of its last message, using the `last` method if available and indexing otherwise. These snippets are therefore very similar, and a decent programmer could immediately tell that they are implementing exactly the same functionality, but they contain enough differences so that a simple token or tree-level comparison fail to actually detect the similarity.

Although we have focused our example on frontend clients, this issue is also very common with

```javascript
function groupMessages(messages) {
  const sortedMessages = messages.sort((a, b) => a.sentAt - b.sentAt);
  const grouped = {};
  for (const message of sortedMessages) {
    if (!grouped[message.senderId]) {
      grouped[message.senderId] = [];
    }
    grouped[message.senderId].push(message);
  }
  const groups = [];
  for (const key in grouped) {
    groups.push(grouped[key]);
  }
  return groups.sort((a, b) => b[b.length - 1].sentAt - a[a.length - 1].sentAt);
}
```

Listing 2.4. Message grouping logic in JavaScript

```java
public static List<List<Message>> groupMessages(List<Message> messages) {
  Collections.sort(messages, (m1, m2) -> m1.getSentAt() - m2.getSentAt());
  HashMap<Integer, List<Message>> grouped = new HashMap<>();
  for (Message m: messages) {
    if (!grouped.containsKey(m.getSenderId())) {
      grouped.put(m.getSenderId(), new ArrayList<Message>());
    }
    grouped.get(m.getSenderId()).add(m);
  }
  List<List<Message>> result = new ArrayList<>(grouped.values());
  Collections.sort(result, (m1, m2) ->
    m2.get(m2.size() - 1).getSentAt() - m1.get(m1.size() - 1).getSentAt()
  );
  return result;
}
```

Listing 2.5. Message grouping logic in Java

```swift
func groupMessages(messages: [Message]) -> [[Message]] {
    let sortedMessages = messages.sorted { (a, b) in a.sentAt <= b.sentAt }
    var groupedMessages = [Int: [Message]]()
    for message in sortedMessages {
        if !groupedMessages.keys.contains(message.senderId) {
            groupedMessages[message.senderId] = [Message]()
        }
        groupedMessages[message.senderId]!.append(message)
    }
    return groupedMessages.values.sorted { (a, b) in
        a.last!.sentAt > b.last!.sentAt
    }
}
```

Listing 2.6. Message grouping logic in Swift

any kind of system where multiple actors consume a same source of information. For example, microservice architecture, where a service can be consumed by a multitude of other services, may also run into the same kind of issue.

2.5 Related work

Many clone detection systems, such as CCFinder [5], DECKARD [7] or SourcererCC [3] support multiple languages but are mostly designed to find clones within the same programming languages and not across them. Some other recent research use deep learning approach for clone detection such as [12] or [13] use deep learning approaches to detect clones within a single programming language, but do not discuss the possibility to extend their method to detecting clones across programming languages. Other approaches such as [14] can work across languages but focus on classifying programs, rather than on detecting code clones. Token-based clone detection tools such as SourcererCC [3] are very efficient to find clones which are introduced by copy-pasting, as these sort of clones are copy-pasted source code share many tokens, but less efficient when clones are introduced by replication of a same functionality by two different entities. When working across programming languages, the chances of two code snippets sharing a large amount of tokens is further reduced, making token comparisons based approaches less likely to detect clones. For example, when comparing listing 2.4 and listing 2.6, line 2 of both listings declare a map-like data structure in order to group messages by sender id. However, this is impossible to infer with simple token comparisons, and would need some encoding about the data types and how they relate to each other to be possible. Some systems such as [15], which are designed to detect clones between programming languages, mitigate this issue by converting the source code to a common AST format, but assumes that the two input languages have a common intermediate representation, in this case both target the .NET platform. A slightly more complicated example is the implementation of the sort function at line 14 in listing 2.4 and line 11 in listing 2.6. In listing 2.4, the access to the last element of the array is done using `a[a.length - 1]`, while in listing 2.6, the same operation is done by using `a.last!`. Although an experienced developer would have no trouble identifying the fact that both codes implement the same functionality, there is currently, to the best of our knowledge, no system capable of detecting such similarities in a completely unsupervised fashion. Finally, an even harder similarity to detect is between lines 10 to 13 in listing 2.4 and the first method call of line 10 in listing 2.6. The purpose of both codes are to get all the values of the map data structure previously created in the form of a list or an array, but although this can be done using the `values` property of the dictionary data structure in Swift, there is no builtin method on the JavaScript object and the same operation therefore requires to loop over its elements. This is an example of a type IV code clone, which up to now no system has managed to detect very accurately — all systems presented here report a recall score close to 0 for type IV clones — and the difference between the languages make this task even harder. However, to find interesting results given the motivating example above, this is one of the types of code clones that we would like to become able to detect.

Chapter 3

Our proposal

In this chapter, we propose a cross-language code clone detection system based on supervised machine learning. Our system can learn to detect clones across programming languages by learning from existing code clones pairs, and is language agnostic in the sense that we use a normalized AST structure which can be generated from any programming language using a parser into feed our system. We also present how to generate the data that our system accepts as input to detect clones, and describe the dataset that we created in order to train and evaluate our system.

3.1 Overview

As discussed in the related works 2.5, current comparison-based approaches to code clone detection do not fit cross-language clone detection, especially when the target languages do not share a common intermediate representation. To overcome this issue, we propose a supervised approach to code clone detection where the system is not given directly information about how code fragments should be compared to each other, but rather uses training data to learn what kind of code fragments should be, or not, considered as clones.

Our system is composed of the following subsystems, which we will describe more thoroughly in the following sections.

1. CC-Learner — Supervised ML model to learn and predict code clones
2. `bigcode-embeddings` — Unsupervised ML model to learn token vector representations
3. `bigcode-astgen` — Parser modules to transform source code to a common AST format
4. `bigcode-ast-tools` — Tools to transform with ASTs generated by `bigcode-astgen`
5. CC-Fetcher — Module to scrape data to train CC-Learner
6. `bigcode-fetcher` — Module to scrape data to train `bigcode-embeddings`

Figure 3.1 shows a general overview of how our system works.

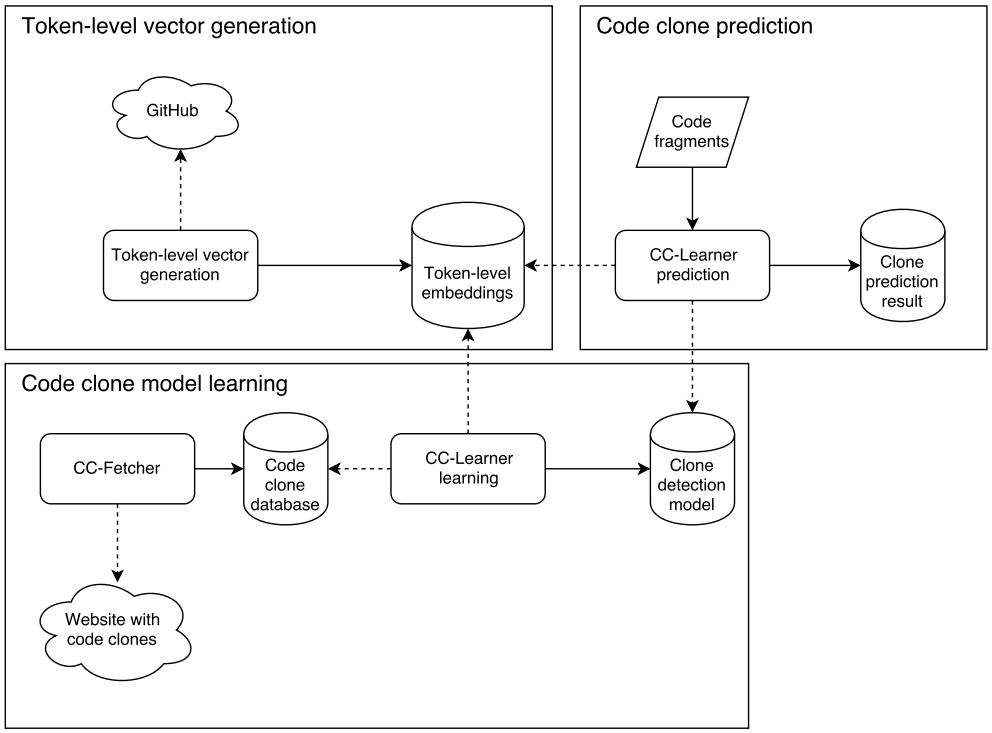

Figure 3.1. Overview of the system

The core of our system is 1, which is an LSTM — presented in 2.3.3 — base supervised machine learning model which takes two code fragments as an input, and predict if the two code fragments are code clones. The flow in training mode is to take two code fragments scraped using the scraper module 5, transform the code pairs to ASTs using the parser modules 3, transform the ASTs in to vectors using the token representation learned by 2 and finally to feed the result to our LSTM-based model 1.

In Section 3.2, we will describe in detail the model we used to detect code clones. In Section 3.3, we will describe the AST generation in order to be able to work with source code written in different programming languages and in Section 3.4, we will explain how we assign a vector in $\mathbb{R}^d$ to each token in the source code.

3.2 Code clone detection model

3.2.1 Model overview

The clone detection model, CC-Learner, that we first introduced in Section 3.1 is the main of our system. Its main role is to actually predict if two code fragments are, or not, clones. The model is composed of the following components.

1. An AST to vector transformer
2. A token-level embedding layer

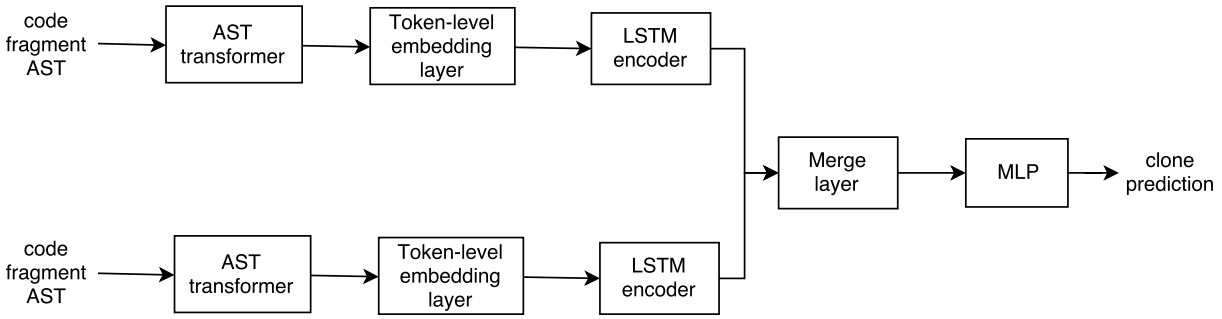

Figure 3.2. CC-Learner code clone detection model

3. An LSTM-based encoder for AST vectors
4. A merge layer to concatenate the two code fragments vectors
5. A feed-forward neural network to make the code clone prediction

Figure 3.2 shows how these parts are connected together. In this section, we will first describe in details the computations done by the model. We will then explain how we train the model and how to use the trained model to predict if two code fragments are clones.

3.2.2 Model computations

We will here formalize the computations each component of the model executes to produce the final prediction.

AST to vector transformer

As input, the model receives an AST, which is a tree structure. The AST to vector transformer takes this AST and maps it into a vector of integers, where each integer represents the index of the token in the AST in the vocabulary of the programming language the source code is written in.

More formally, if we let $\mathcal{T}_l$ be the set of all possible ASTs for source code written in programming language l, the AST transformer f_t^l will be a function with the following definition.

$$f_t^l : \mathcal{T}_l \to \mathbb{N}^m \tag{3.1}$$

where m may vary depending on the size of the input tree and the algorithm used for f_t^l. Furthermore, given V_l is the vocabulary of tokens for the programming language l, the following equation must hold for any implementation of f_t^l.

$$\forall t \in \mathcal{T}_l, n \in f_t^l(t),\ n \in V_l \tag{3.2}$$

For CC-Learner, we provide two different instances of this function, and the function which is used is a hyper parameter of our model. We will use the code snippet 3.1, the AST of figure 3.3 and the vocabulary of table 3.1 to illustrate how both of our instances work.

```
1 if (a < 1) {
2   return 0;
3 }
```

Listing 3.1. Sample code snippet

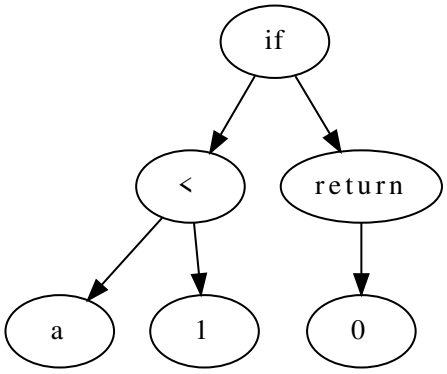

Figure 3.3. Sample AST

Table 3.1. Sample vocabulary

Token	Index
`if`	1
identifier	2
integer	3
`<`	4
`return`	5

The first instance of our AST to vector transformer, f_{t_1} is very simple. The idea is simply to traverse the tree using a depth-first, pre-order tree traversal. At each node in the tree, we look up the index of the node token in the vocabulary, and we append the index to the output vector.

In our example above, the algorithm will start at the **`if`** node, it will lookup the index of the node in the vocabulary, which in this case is 1, and append 1 to the output vector. It will then do the same thing with the other nodes in the following order: `<`, `a`, `1`, **`return`**, `0`. After completion, we will therefore obtain the following vector in $\mathbb{N}^6$.

$$\begin{pmatrix} 1 & 4 & 2 & 3 & 5 & 3 \end{pmatrix} \tag{3.3}$$

When using this algorithm, the output dimension is always the same as the number of nodes in the input AST.

Although the first instance of our AST to vector transformer is very simple and straightforward, the main disadvantage is that it completely loses information about the topology of the input tree. To mitigate this issue, we also implemented another function f_{t_2} which tries to partially preserve the topology of the tree in the output vector. In the above example, **`if`** has two children nodes: `<` and **`return`**, therefore the distance inside the tree topology between **`if`** and both of these nodes is 1. However, in the output vector, the distance between **`if`** and `<` is 1, but the distance between **`if`** and **`return`** is 4. For such a simple case, the distance is still short enough for the neural to be able to capture relationship between these elements, but in more complex cases, a distance of 1 inside the tree topology could easily become extremely large. This is the main issue we try to tackle with our other instance of the AST to vector transformer function. The idea is to perform a depth-first, pre-order traversal of the tree and generate the same vector as in the first instance, but also to traverse the tree once more this time using a depth-first, pre-order traversal but selecting the nodes from right to left instead of selecting them from left to right when executing the traversal. Using the above example, this means that in the second traversal, after visiting **`if`**, instead of visiting `<`, we visit **`return`**. The second traversal order would therefore be **`if`**, **`return`**, `0`, `<`, `1`, `a`. Once we have finished traversing the AST in both direction, we concatenate the vectors obtain by each traversal to obtain the final vector for the AST. For the above example, the final vector will be

$$\begin{pmatrix} 1 & 4 & 2 & 3 & 5 & 3 & 1 & 5 & 3 & 4 & 3 & 2 \end{pmatrix}$$

As we are concatenating two vectors generated by a full traversal of the tree, and which will therefore each have the dimension of the number of nodes in the tree, the final output vector will have twice the dimension of the number of nodes in the tree. Therefore, if there are n nodes in the tree, the output will be a vector in $\mathbb{N}^{2n}$.

Token-level embedding layer

When we transform an AST in to a vector using the above approach, we obtain a vector of indexes. As each index cannot be fed directly to the encoder, the two main approaches are to use a one-hot vector for the index, or to use a distributed vector representation. As using a distributed vector representation has many advantages [9] over using one-hot vectors, we choose this approach for our model. We will here assume that we already have the embedding for each token available, and will describe how we actually compute these embedding in Section 3.4.

Table 3.2. Sample embedding of dimension 3

Index	Embedding
1	$\begin{pmatrix}0.12 & -0.43 & 0.66\end{pmatrix}$
2	$\begin{pmatrix}-0.81 & 0.01 & 0.28\end{pmatrix}$
3	$\begin{pmatrix}0.91 & 0.33 & 0.47\end{pmatrix}$
4	$\begin{pmatrix}-0.51 & -0.27 & -0.09\end{pmatrix}$
5	$\begin{pmatrix}0.17 & 0.73 & -0.56\end{pmatrix}$

The embedding layer of the model transforms a vector in $\mathbb{N}^n$ into a matrix in $\mathbb{R}^{n\times d_w}$ where d_w is the dimension of the embedding and is a hyper parameter of the model. The embedding layer for a programming language l is therefore a function f_w^l with the following dimensions.

$$f_w^l : \mathbb{N}^n \to \mathbb{R}^{n\times d_w} \tag{3.4}$$

Such embedding are actually represented by a matrix W^l which has dimensions of $\mathbb{R}^{|V|\times d_w}$ where $|V|$ is the size programming language l vocabulary, and d_w is the dimension of the embedding. Sample values for embedding of dimension $d_w = 3$ are show in table 3.1. The sample embedding given in table 3.1 are shown in their matrix form in equation 3.5.

$$W^l = \begin{pmatrix} 0.12 & -0.43 & 0.66 \\ -0.81 & 0.01 & 0.28 \\ 0.91 & 0.33 & 0.47 \\ -0.51 & -0.27 & -0.09 \\ 0.17 & 0.73 & -0.56 \end{pmatrix} \tag{3.5}$$

Transforming an index i in its vector representation therefore reduces to selecting the ith index of the matrix W^l, and the function f_w can therefore be expressed as in equation 3.6, and is computationally inexpensive. In 3.6, W_i means selecting the ith row of the matrix W.

$$f_w^l(i) = W_i^l \tag{3.6}$$

Given the embedding of dimension $d_w = 3$ given in table 3.2 for the sample vocabulary presented in table 3.1, the vector of indexes in equation 3.3 will be transformed in the matrix of equation 3.7.

$$\begin{pmatrix} 0.12 & -0.43 & 0.66 \\ -0.51 & -0.27 & -0.09 \\ -0.81 & 0.01 & 0.28 \\ 0.91 & 0.33 & 0.47 \\ 0.17 & 0.73 & -0.56 \\ 0.91 & 0.33 & 0.47 \end{pmatrix} \tag{3.7}$$

In the matrix shown in equation 3.7, each row is the embedding of the index in the original vector.

LSTM-based encoder

As discussed previously, the output of the embedding layer will be a matrix of dimension $\mathbb{R}^{n \times d_w}$ where n depends on the size of the input AST and d_w is the dimension of the embedding, which is a hyper-parameter of the model. The purpose of the LSTM-based encoder layer is to transform this matrix into a vector in d_e, where d_e is a hyper-parameter of the model, which captures the relation between the elements in the input matrix. This task is the equivalent of transforming a matrix representing a sentence to a vector representing the same sentence for natural language processing. This layer will therefore be a function f_e^l defined as in equation 3.8.

$$f_e^l : \mathbb{R}^{n \times d} \to \mathbb{R}^{d_e} \tag{3.8}$$

An interesting property of this function is that the output dimension is independent of the input dimension, which makes it possible to easily aggregate inputs which originally had different dimensions in the next layer of the model.

To achieve this task, we use an LSTM model as described in 2.3.3, and feed it the result of the embedding layer. d_e is chosen experimentally, and is usually between 20 and 200 depending on the other hyper-parameters of the model.

Given the result shown in equation 3.7 and $d_e = 4$, equation 3.9 shows a sample output of the LSTM encoder.

$$\begin{pmatrix} 0.81 & -0.12 & 0.77 & -0.45 \end{pmatrix} \tag{3.9}$$

There are also other choices that need to be made when instantiating the LSTM. We first need to decide if the LSTM will be bi-directional or not. When the LSTM is bi-directional, there will actually be two LSTMs: an LSTM for the forward direction of the output of the embedding layer, and an LSTM for the reverse direction. After the two computations run, the result of the two LSTMs will be concatenated to give the final output. The final dimension of the vector will therefore be $2d_e$ instead of d_e. Equation 3.10 shows a sample output for equation 3.7 as input and $d_e = 4$ when using a bi-directional LSTM.

$$\begin{pmatrix} 0.81 & -0.12 & 0.77 & -0.45 & -0.68 & 0.22 & 0.94 & -0.03 \end{pmatrix} \tag{3.10}$$

Another decision to make for the LSTM-based encoder is whether to use a single LSTM which works as described in 2.3.3, or to stack multiple LSTMs above one another. When stacking multiple LSTMs, the first LSTM works exactly a before, however, its output is not directly used but fed to another LSTM and only the uppermost LSTM result is used for the next step. We try to use 1 to 3 stacked LSTM and chose the best value for our use case experimentally.

Merge layer

Up to know, we have focused on how to encode a single AST into a vector in $\mathbb{R}^{d_e}$. However, our model takes two inputs, and produces a single output. We therefore need to combine these two

inputs in some way that allows us to generate a single prediction to whether or not the two given inputs are code clones or not. This is the role of merge layer in our model. It takes two inputs and return a single output. The merge layer is therefore defined by a function f_m as shown in equation 3.11, where n normally depends on d_e.

$$f_m : \mathbb{R}^{d_e} \times \mathbb{R}^{d_e} \to \mathbb{R}^n \tag{3.11}$$

In our model, we define two different types of merge layers which we will describe below.

The first instance of the merge layer, f_{m_1}, is used as a baseline, and is a simple concatenation operation. We take the vector outputted by the LSTM-based encoder for the two input ASTs, and concatenate them into a single vector in $\mathbb{R}^{2d_e}$. In this case, we therefore have $n = 2d_e$. For example, given the LSTM output given in equation 3.9 and another LSTM output given by equation 3.12,

$$\begin{pmatrix} 0.12 & 0.86 & -0.27 & 0.33 \end{pmatrix} \tag{3.12}$$

the merge layer would output the following vector.

$$\begin{pmatrix} 0.81 & -0.12 & 0.77 & -0.45 & 0.12 & 0.86 & -0.27 & 0.33 \end{pmatrix} \tag{3.13}$$

The second instance of the merge layer, f_{m_2}, uses the combination of the angle and the distance of the two vectors described in [16]. Given h_L the output of the encoder for the first code snippet and h_R the output of the encoder for the second code snippet, the output h of the merge layer is defined by equation 3.14.

$$\begin{aligned} h_\times &= h_L \odot h_R \\ h_+ &= |h_L - h_R| \\ h &= W^{(\times)} h_\times + W^{(+)} h_+ \end{aligned} \tag{3.14}$$

When using this merge strategy, $W^{(\times)}$ and $W^{(+)}$ are parameters of the model and are both matrices with dimension $\mathbb{R}^{d_o \times d_e}$ where d_o is a hyper-parameter of the model. The dimensions of the different results will therefore be $h_\times \in \mathbb{R}^{d_e}$, $h_+ \in \mathbb{R}^{d_e}$ and $h \in \mathbb{R}^{d_o}$. Using the outputs given in equations 3.9 and 3.12, we will therefore obtain the following results.

$$\begin{aligned} h_\times &= \begin{pmatrix} 0.0972 & -0.1032 & -0.2079 & -0.1485 \end{pmatrix} \\ h_+ &= \begin{pmatrix} 0.69 & 0.98 & 1.04 & 0.78 \end{pmatrix} \end{aligned}$$

h will then be obtain by doing a dot product using the weights of the model and the results shown above.

Feed-forward neural network

Once we obtain a single vector containing information about the two input code snippets, we use a feed-forward neural network to predict if the inputs were code clones or not. The output layer of our neural network uses a sigmoid function defined as in 2.16, and therefore outputs a real number between 0 and 1 which can be interpreted as a probability.

The feed-forward neural network can therefore be represented by a function f_p defined as follow.

$$f_p : \mathbb{R}^{d_o} \to [0, 1] \tag{3.15}$$

This layer has the usual hyper-parameters found in feed-forward neural networks, for example, the number of hidden layers, the number of units per layer and the number or the activation function of the hidden layers. We choose these hyper-parameters experimentally.

Model training

We will now describe how the weights of the model defined using the layers described in this section can be trained. First, using the function definitions above, we can define our model using the following functions.

$$f_E^l(x) = (f_e^l \circ f_w^l \circ f_t^l)(x) \tag{3.16}$$

$$f^{(l_1,l_2)}(x, y) = f_p\left(f_m\left(f_E^{l_1}(x), f_E^{l_2}(y)\right)\right) \tag{3.17}$$

In 3.17, x is a code snippet AST written in programming language l_1 and y is written in programming language l_2. To be able to train the model, we first need to define a loss function for our model. As we output a probability, we choose the binary cross entropy loss function defined as follow.

$$L(y, \hat{y}) = y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \tag{3.18}$$

In equation 3.18, y is the actual output — 1 if the two input code snippets are clones and 0 otherwise — and $\hat{y}$ is the value predicted by our model. Our loss for a sample can therefore be computed using the following equation.

$$E(x_1, x_2, y) = L(y, f(x_1, x_2)) \tag{3.19}$$

In order to train our model, we compute the gradient of the our error E with respect to the weights $\mathbf{W}$ of the model, $\partial E/\partial \mathbf{W}$ using back-propagation, and then apply an optimization algorithm such as gradient descent or Adam [17].

To execute the training, we simply need to have many samples of code pairs, with a label telling us if the pair is, or not, a code clone. We can then use these sample to feed our model, back-propagate the error and update the weights of the model as for any other machine learning model.

```
1 def add(a, b):
2     return a + b
```

Listing 3.2. `add` function in Python

```
1  [{"id": 0, "type": "Module", "children": [1]},
2     {"id": 1, "type": "FunctionDef", "value": "add", "children": [2, 5]},
3       {"id": 2, "type": "arguments", "children": [3, 4]},
4         {"id": 3, "type": "arg", "value": "a"},
5         {"id": 4, "type": "arg", "value": "b"},
6       {"id": 5, "type": "body", "children": [6]},
7         {"id": 6, "type": "Return", "children": [7]},
8           {"id": 7, "type": "BinOpAdd", "children": [8, 9]},
9             {"id": 8, "type": "NameLoad", "value": "a"},
10            {"id": 9, "type": "NameLoad", "value": "b"}]
```

Listing 3.3. add function JSON AST

3.3 AST generation

In order to be able to work with source code written in different languages, we implemented multiple AST generation tools which generate ASTs in a common format for different programming languages. The tool we developed is available on GitHub[1] and supports source code written in Python, Java and JavaScript.

As a common format for ASTs, we chose to use the JSON format described in [18]. A JSON AST is a JSON array of objects, where each object contains the following properties.

- `id` (**`int`**, required) — unique integer identifying current AST node
- `type` (**`string`**, required) — type of current AST node
- `value` (**`string`**, optional) — value of current AST node if any
- `children` (**`list[int]`**, optional) — children **`id`**s of current AST node if any

Another thing to note about this format is that the objects in the list follow the order obtained when doing a pre-order, depth-first search on the AST. Although this is not a constraint, keeping such an order allow to have a simpler algorithm to reconstruct a real tree-structured AST from the JSON formatted AST. Depending on the language, we would possible need to have other properties to express completely the source code, we choose to focus solely on the properties listed above to avoid keeping language-specific information. To illustrate how the AST is encoded into JSON, listing 3.2 is a simple Python function that adds two numbers, and listing 3.3 is its JSON AST representation.

[1] `https://github.com/tuvistavie/bigcode-tools/tree/master/bigcode-astgen`

We indented the JSON to make the object more readable, but it is a simple JSON array and has therefore no indent or nesting property at a structural level. The nesting of the child is done by having the indexes of the children in the `children` property of each object.

To generate the ASTs in each language, we used the JavaParser[2] library for Java, the `ast` module[3] for Python and the Acorn [4] library for JavaScript.

To perform most tasks, we need to restore a tree structured from the flat JSON formatted AST. Algorithm 3.1 describes a recursive algorithm to perform this task.

Algorithm 3.1 Tree-structured AST from JSON formatted AST

```
1: function CreateAST(jsonArray)
2:     return ASTFromJSONObject(jsonArray, 0)
3: end function
4: function ASTFromJSONObject(jsonArray, nodeId)
5:     jsonNode ← jsonArray[nodeId]
6:     node ← new Node
7:     node.id ← jsonNode.id
8:     node.type ← jsonNode.type
9:     node.value ← jsonNode.value
10:    node.children ← []
11:    for childId in jsonNode.children do
12:        childNode ← ASTFromJSONObject(jsonArray, childId)
13:        node.children ←+ childNode
14:    end for
15:    return node
16: end function
```

As mentioned above, we assume the JSON array is generated using a pre-order, depth-first search. We therefore know that the root of the AST will be the first element of the array. `CreateAST` therefore delegates the creation of the AST to `ASTFromJSONObject`, which will receives the root and recurse from it to generate the full tree. `ASTFromJSONObject` will assign all the properties from the JSON node it received to a new node object, except for the children. It enumerates on each child id in the JSON node children array, and recursively calls itself, passing in the child index as its second parameter.

3.4 Token-level vector representation

As discussed previously, we do not feed our model one-hot vector representation of the tokens, but rather token embedding. In this section, we explain how we generate token-level vector representation, also known as embedding, for each token in the vocabulary of a programming

[2] `http://javaparser.org/`
[3] `https://docs.python.org/3/library/ast.html`
[4] `https://github.com/acornjs/acorn`

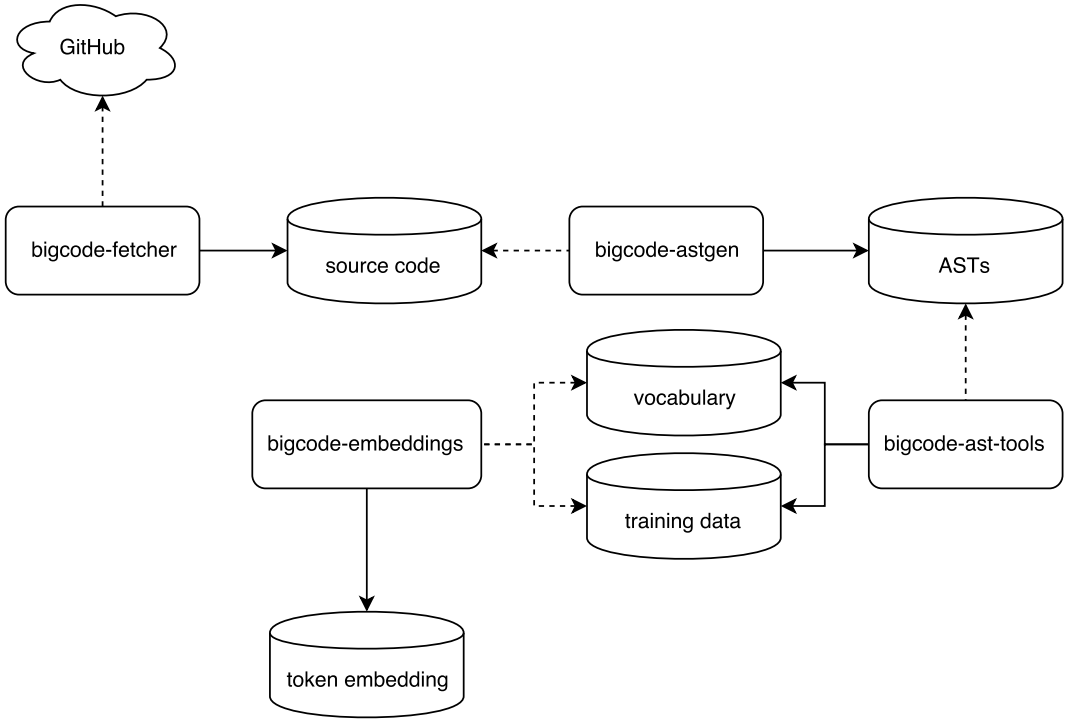

Figure 3.4. Embedding generation

language. The process for generating token embedding for a programming language l can be divided in the following steps.

1. Collect source code written in l
2. Generate the vocabulary of l
3. Generate data to train a skipgram model
4. Train a skipgram model using the data generated in 2.

We show an overview of the token-level embedding generation process in figure 3.4. In the rest of this section, we will describe the different algorithms to generate the vocabulary, generate the data to train a skipgram model, as well as the details of the skipgram model. We will leave the source code collection details for Chapter 4.

3.4.1 Vocabulary generation

To be able to learn embedding for different tokens in a target programming language, we first generate a finite set of tokens — the vocabulary of the programming language. There are a few different choices to consider when generating the vocabulary of the language, the main choices we consider are:

```
1 def is_minor(age):
2     if age >= 18:
3         print("you␣are␣major")
4     else:
5         print("you␣are␣minor")
```

Listing 3.4. `is_minor` function in Python

1. Should application specific tokens be included in the vocabulary?
2. How large should be the vocabulary?

We define application specific tokens as tokens that can be unique to a program, for example, identifiers such as variable and method names or values in literals. If we choose to ignore the application specific tokens, we do not remove the token but rather ignore its value and keep only its type. We will consider the code snippet in listing 3.4 to explain how we can treat application specific tokens.

First, given we take all application specific tokens, we will have the following tokens.

```
def, is_minor, age, if, age, >=, 18, print, "you␣are␣major",
else, "you␣are␣minor"
```

On the other hand, if we decide to strip all the application specific tokens, we will replace all the token with a value by their type, and therefore all identifiers will be replaced by a common `<ident>` token, all string literals will be replaced by a common `<string>` token, and only the following tokens set will remain in the vocabulary.

```
def, <ident>, if, <binary_op>, <string>, else
```

If we choose to ignore all application specific tokens, we only keep a predictable number of tokens that the target programming language defined in its specifications, and we can therefore ignore the size of the vocabulary issue altogether as the number of tokens we might expect will always be small enough.

When using application specific tokens, although we could simply use all the tokens as shown in the example above, all tokens might not be very significant and we therefore need to choose what tokens we want to keep and what tokens we want to ignore. As many identifiers can be found across applications, especially the one found in the standard library or well-known libraries of a programming language, we decide to keep all identifiers. However, we decide not to keep string literals as these are too rarely shared between different programs. Another issue when using application specific tokens is the case where a token encountered after having generated our vocabulary does not exist in the vocabulary, in which case it is not possible to map it directly to an index in the vocabulary. A common approach to this issue in natural language processing is to assign a special index which is used for all unknown tokens. However, in our case, each token has a type and a value, we therefore decide to use the type without its value if the type and value pair of the token cannot be found in the vocabulary. This means that for every type in the target programming language, we will have a token representing the type with no value in

the vocabulary. When looking up the token in the vocabulary, we first look for a token with the same value and type. If we find one, we use its index, otherwise we fallback to looking for the token with the same type instead and use its index. As there are only a limited number of token types in a programming language, we are therefore sure to at least find the matching type.

When using application specific tokens, the number of tokens will increase with the number of files used to generate the vocabulary, and we therefore need to set a threshold to the number of tokens we want to use. This threshold is set experimentally, and given n as this threshold, our vocabulary will be composed of the n most common tokens from the files that we used to generate the vocabulary.

In algorithm 3.2, we give a high-level overview of our vocabulary generation algorithm and in algorithm 3.3 we show how we lookup a token in the vocabulary.

Algorithm 3.2 Vocabulary generation algorithm

```
1: function GenerateVocabulary(sourceFiles, includeValues, maxSize)
2:     tokensCount ← empty map
3:     for file in sourceFiles do
4:         tokens ← tokenize(file)
5:         for token in tokens do
6:             if (token.type, token.value) ∉ tokensCount then
7:                 tokensCount[(token.type, token.value)] ← 0
8:             end if
9:             Increment tokensCount[(token.type, token.value)]
10:            if includeValues then
11:                if (token.type, null) ∉ tokensCount then
12:                    tokensCount[(token.type, null)] ← 0
13:                end if
14:                Increment tokensCount[(token.type, null)]
15:            end if
16:        end for
17:    end for
18:    Sort tokensCount by value
19:    vocabulary ← first maxSize keys of tokensCount
20:    return vocabulary
21: end function
```

Algorithm 3.3 Token lookup

```
1: function LookupTokenIndex(vocabulary, token)
2:     if (token.type, token.value) ∈ vocabulary then
3:         return vocabulary.indexOf((token.type, token.value))
4:     end if
5:     return vocabulary.indexOf((token.type, null))
6: end function
```

In algorithm 3.2, `sourceFiles` is a list of source code files to use to generate the vocabulary, `includeValues` is a boolean that is **`true`** if we want to use token values and **`false`** if we do not — i.e. if we do not use anything application specific. `maxSize` is the maximum number of tokens in the generated vocabulary. In algorithm 3.3, `vocabulary` is the vocabulary generated by `GenerateVocabulary` and `token` is the token that needs to be lookup. It is worth noting that for algorithms relying on this method to work in a reasonable time, `indexOf` must run in constant time $\mathcal{O}(1)$, and is therefore implemented using a hash.

3.4.2 Generating data to train skipgram model

After generating the vocabulary, we need to generate data to train a skipgram model, presented in 2.3.2. In the context of natural language processing, the input is usually considered as a sequence, and the context of a particular word is the words before and after this word in the sequence of words used for training. Furthermore, the distance between the word and its context is normally context by a single window size hyper-parameter. We could apply the same approach by treating a program as a sequence of tokens, but we decide to take advantage of the topological information contained by the AST, we decide to work on the AST rather than on a simple sequence of token. We therefore need to define the context of a word, or rather a token in our case, differently than for a sequence.

In the context of a tree, a node is directly connected to its parent and its children. We can therefore define parents and children to be the context of a node. Depending on the use case, the siblings of a node could also be viewed as a viable candidate its context. A single window size parameter could be used to control how deep upward and downward should the context of a node be. However, while a node will only have a single parent, it can have any number of children, and therefore, while having a window size of 3 for the ancestors would only generate 3 nodes in the context, if every descendant of a node had 5 children, a window size of 3 would generate $5^3 = 125$ nodes in the context, which would probably generate more noise than signal when trying to train a skipgram model. We therefore decide to use two different parameters to control the window size of the ancestors and the window size of the descendant when generating the data to train our skipgram model. When we do include siblings in the context, we decide to only use the direct siblings of the nodes and not to use the siblings of the ancestors, although this could also be another parameter of the algorithm. In algorithms 3.4, 3.5 and 3.6, we describe the algorithms we use to generate the data to train a skipgram model.

Algorithm 3.4 takes as input a list of files written in the programming language for which we want to generate token embedding, the vocabulary extracted for this programming language and the parameters described above. It loops over all the nodes in the file, and uses algorithms 3.5 and 3.6 to find all nodes in the context of the current node, and returns a list of pair of indexes where each pair represent a target node and a node in its context. Algorithm 3.5 takes as input a node and the parameters described above, and returns the set of nodes in the context of the given node. It first uses algorithm 3.6 to find all the descendants of the node in the window given by the passed parameters, then find all the ancestors in the given window and finally add the siblings to the set of results if necessary. Algorithm 3.6 takes a node, the maximum depth up to which descendants should be populated and the current depth — which will initially be set to 0 — and returns the set of descendants up to the passed maximum depth for the node. It

Algorithm 3.4 Data generation for skipgram model

```
1: function GenerateSkipgramData(files, vocabulary, params)
2:     skipgramData ← {}
3:     for file in files do
4:         ast ← GenerateAST(file)
5:         for node in ast do
6:             nodeIndex ← LookupTokenIndex(node)
7:             contextNodes ← GenerateContext(node, params)
8:             for contextNode in contextNodes do
9:                 contextNodeIndex ← LookupTokenIndex(contextNode)
10:                skipgramData ←+ (nodeIndex, contextNodeIndex)
11:            end for
12:        end for
13:    end for
14:    return skipgramData
15: end function
```

Algorithm 3.5 Context generation for an AST node

```
1: function GenerateContext(node, params)
2:     contextNodes ← FindDescendants(node, params.descendantWindowSize, 0)
3:     parent ← node.parent
4:     n ← 0
5:     while parent is defined ∧ n < params.ancestorWindowSize do
6:         contextNodes ←+ parent
7:         parent ← parent.parent
8:         n ← n + 1
9:     end while
10:    if params.includeSiblings ∧ node.parent is defined then
11:        for sibling in node.parent.children do
12:            contextNodes ←+ sibling
13:        end for
14:    end if
15:    return contextNodes
16: end function
```

Algorithm 3.6 Find descendants for a node until given depth

```
1: function FindDescendants(node, maxDepth, currentDepth)
2:     if currentDepth ≥ maxDepth then
3:         return {}
4:     end if
5:     descendants ← node.children
6:     for child in node.children do
7:         childDescendants ← FindDescendants(child, maxDepth, currentDepth+1)
8:         descendants ← descendants ∪ childDescendants
9:     end for
10:    return children
11: end function
```

first adds all the children of the current node to the set of descendants, then recurses through all the children until the current depth is equal to the maximum depth for which to generate descendants.

3.4.3 Training a skipgram model

Once the data is generated using algorithms 3.4, 3.5 and 3.6, the last step needed to generate embedding is to actually train a skipgram model using the generated data. Algorithm 3.4 generates pairs of indexes which can be directly fed to a neural network, and there is therefore no need for further pre-processing. To train the model, the vocabulary used is the same as the one used to generate the skipgram data and the size of the embedding is a hyper parameter of the model. The model is trained using a negative sampling objective shown in equation 3.20 as given in [9].

$$\log \sigma \left({v'_{w_O}}^T v_{w_I} \right) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma \left({v'_{w_i}}^T v_{w_I} \right) \right] \tag{3.20}$$

In equation 3.20, v are the weights of the hidden layer, which will be used as embedding, v' are the weights of the output layer, w_I is an input token and w_O is a token in its context. w_i are the "noise" tokens which are randomly sampled from the vocabulary using a noise distribution P_n, for which we use a unigram distribution for reasons described in [9]. The dimension of v is $\mathbb{R}^{|V| \times d}$ where $|V|$ is the size of the vocabulary and d is the size of the embedding, which is a hyper-parameter of the model.

Chapter 4

Experiments

In this chapter, we will describe the experiments we made and present the results we obtained. In Section 4.1, we will present our experiments about code clone detection, while in Section 4.2 we will describe our experiments to generate token embedding. In both sections we will show what kind of data we used to train our model and how our model perform with different hyper-parameters.

To run our experiments, we used the two machines shown in table 4.1. In table 4.2, we show the software we used to implement and run our machine-learning related experiments and the version we used.

4.1 Code clone detection

To evaluate our clone detection model, we implemented the model we described in 3.2, we collected data of programs implementing the same functionality to feed to the model, and we trained and evaluated our model with different hyper-parameters. In this section, we will give some details about the data we collected for our experiments and present the different results we obtained when evaluating our model.

Table 4.1. Machines used for experiments

	Machine 1	Machine 2
CPU	Intel i7-6850K	Intel Xeon E5-2637 v3
CPU frequency	3.60GHz	3.50GHz
Cores count	6	8
Threads count	12	16
Memory	64GB	512GB
GPU	Nvidia Quadro P6000	Nvidia Tesla P100
GPU memory	24GB	16GB

Table 4.2. Software used for experiments

Software	Version
Linux	4.14 / 4.4
CUDA	8.0
cuDNN	6.0
Python	3.6.2
Tensorflow	1.3.1
Keras	2.1.2

4.1.1 Code clone detection dataset

As we are using a supervised learning approach to code clone detection, to train our model we needed data which fulfills the following properties.

1. Multiple code fragments should implement the same functionality
2. Information on whether two code fragments implement the same functionality must be included
3. Dataset should contain code fragments written in at least 2 programming languages

To the best of our knowledge, no dataset currently available fulfills all the necessary properties to our experiments, therefore we created our own dataset.

For this dataset, we found that competitive programming websites are an excellent match. The solution to a single problem is implemented by a large number of persons in many different languages. All the solutions to a single problem must implement exactly the same functionality, therefore, we are assured that all source codes implementing a solution to the same problem are at least type IV code clones. The easier the problem is, the higher the probability of code fragments implementing the solution to the same problem have to be very similar to each other, and to therefore be closer to type III clones. Furthermore, multiple solutions to a problem are always implemented by different users, which makes our dataset closer to the motivating example we presented in Section 2.4.

To create the dataset, we scraped code from a famous Japanese competitive programming website. As our implementation currently only supports Java and Python, we fetched data for these two programming languages. We restricted the data to only programs that were accepted by the website judging system — meaning that the programs actually implemented the solution to the given problem — in order to have the type IV code clone guarantee. In table 4.3, we give some statistics about the dataset we created and in table 4.4 we show how the given statistics are distributed between Python and Java. We show the distribution of the number of files per problem — which is representative of the number of clones in the dataset — in figure 4.1.

In order to be able to feed the code to our model, we transform the code we fetched using `bigcode-astgen` we presented in Section 3.3 using the commands shown in listing 4.1.

Table 4.3. Clone detection dataset overview

Measure	Value
Problems count	576
Avg. files / problem	77
Files count	44620
Lines count	1270599
Tokens count	8554476

Table 4.4. Clone detection dataset language overview

	Python	Java
Avg. files / problem	41	36
Files count	23792	20828
Lines count	312353	958246
Tokens count	1810085	6744391

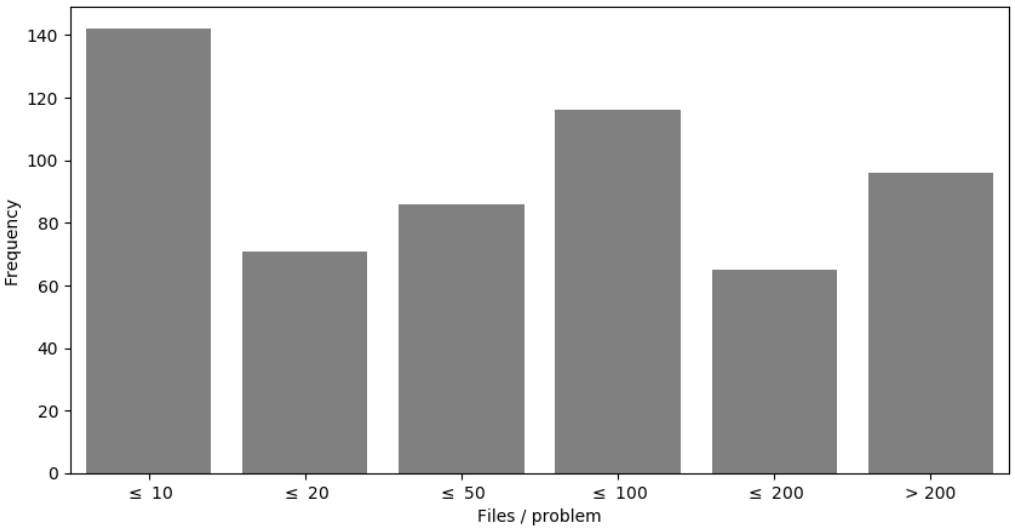

Figure 4.1. Distribution of the number of files per problem

```
bigcode-astgen-java --batch -f 'src/**/*.java' -o java-asts
bigcode-astgen-py --batch -f 'src/**/*.py' -o python-asts
cat java-asts.json python-asts.json > asts.json
cat java-asts.txt python-asts.txt > asts.txt
```

Listing 4.1. AST generation using `bigcode-astgen`

With the first two commands, we generate the ASTs for Python and Java source files, and with the next we commands we merge the ASTs and the file which maps the index of the AST in the JSON file to the name of the file in the original dataset.

4.1.2 Code clone detection model hyper-parameters

Our model contains many hyper-parameters that can vastly influence its performances while detecting clones. In listing 4.2, we show a sample configuration file that we actually use to train our model. The file is written using in YAML and is loaded by our system when training and evaluating the model. Although the file we show here is not complete, it contains the main settings that one might want to change when training a model.

We describe the meaning of each parameter in our configuration file in table 4.5.

Although there is a very large number of hyper-parameters, not all affect our model in the same way. For example, while the vocabulary we use, or the way we transform the AST into a vector affects greatly the performance of the model, other parameters such as the optimizer we use or the batch size do not are not as important. We will explain more in detail how each section in our configuration is used, and explain some of the decisions to be made when setting these parameters.

The parameters under each object of the `languages` section contains the necessary information to encode an AST into a vector for the given programming language. The vocabulary and embedding we pass will determine how each token is encoded into its vector representation. We will discuss further how these embedding are trained in Section 4.2, but the most important decision being made for the embedding is if we want to use the values of the tokens or simply their types, resulting in a small vocabulary in the order of 100 tokens in the first case, or a much larger vocabulary of thousands of tokens in the second. The input length is the maximum number of tokens for a single code fragment. We use this limit mainly for performance purpose, as we pad all the inputs to this size to be able to train our model using batches. Given our current implementation, removing this limits forces us to train the model a sample at a time, largely hurting performance. This shortcoming can be mitigated with techniques such as bucketing [19], but we leave this for future work. The `output_dimensions` and `bidirectional_encoding` parameters control what kind of LSTM network will be used to encode the vectorized AST into a single vector.

Other parameters under `model` contain hyper-parameters on how the input vectors should be merged, as well as what kind of network to use to actually make the prediction. It also contains information about the optimizer to use, and may have hyper-parameters for the optimizer such as its learning rate.

The part under `generator` contains all the information necessary to generate data to train

```yaml
references:
  language_defaults: &language_defaults
    embeddings_dimension: 50
    output_dimensions: [30]
    transformer_class_name: BiDFSTransformer
    bidirectional_encoding: false

model:
  languages:
    - name: java
      <<: *language_defaults
      input_length: 800
      vocabulary: /path/to/java-vocab.csv
      embeddings: /path/to/java-embeddings.csv
    - name: python
      <<: *language_defaults
      input_length: 800
      vocabulary: /path/to/python-vocab.csv
      embeddings: /path/to/python-embeddings.csv

  merge_mode: bidistance
  merge_output_dim: 64
  dense_layers: [64]
  optimizer:
    type: rmsprop

generator:
  submissions_path: /path/to/submissions.json
  asts_path: /path/to/asts.json
  split_ratio: [0.8, 0.1, 0.1]
  shuffle: true
  negative_samples: 4
  negative_sample_distance: 0.2
  class_weights: {0: 2.0, 1: 1.0}

trainer:
  batch_size: 128
  epochs: 20
```

Listing 4.2. Model configuration file

Table 4.5. Clone detection model hyper-parameters

Parameter	Notation	Description	Sample value
`transformer_class_name`	f_t	The algorithm to used to encode the AST into a vector as described in 3.2.2	`DFSTransformer`
`vocabulary`		The path to the extracted vocabulary	`vocab.tsv`
`vocabulary_size`	$\|V\|$	The size of the vocabulary	500
`embeddings`		The path to the trained token embedding. If this parameter is not provided, embedding are randomly initialized	`embeddings.pickle`
`embeddings_dimension`	d_w	Dimension of the embedding for target language	100
`output_dimensions`	d_e	Dimension of the LSTM encoder. If the size is greater than 1, the LSTM will be stacked and the output dimension will be the value of the last element of the list	`[100, 50]`
`bidirectional_encoding`	f_{e_b}	Whether or not to use a bidirectional LSTM	**`true`**
`name`		The name of the programming language	java
`input_length`	$\|X\|$	The max. number of tokens per code fragment. This restriction can be avoided at the cost of a performance penalty	1000
`merge_mode`	f_m	Algorithm to use to merge the two code fragments as described in 3.2.2	bidistance

`merge_output_dim`	d_o	Output dimension to use for the merge layer. Currently only effective when using equation 3.14 as the merge layer	`50`
`dense_layers`	$d(f_p)$	The number of hidden layers to use for the final MLP and their number of units	`[50, 50]`
`optimizer`		The optimizer to use to train the model	rmsprop
`submissions_path`		The path of the submissions metadata	`submissions.json`
`asts_path`		The path of the training data ASTs generated as described above	`asts.json`
`split_ratio`		The amount of data to use for training, hyper-parameter tuning, and testing the model	`[0.8, 0.1, 0.1]`
`shuffle`		Whether or not to shuffle the data when training the model	**`true`**
`negative_samples`	$\|S_n\|$	The number of non-clone noise data — negative sample — to generate per code clone	4
`negative_sample_distance`	ΔS_n	The maximum distance ratio between the size of the input and the size of the negative sample, to avoid the model to try to detect clones by number of tokens	0.2
`class_weights`	L_w	The weight to assign to positive and negative samples. The higher the weight, the more an error will be penalized	`{0: 2.0, 1: 1.0}`
`batch_size`		The size of a batch when training the model	256
`epochs`		The number of epochs to train the model	10

Table 4.6. Java/Python clone detection models parameters

n	$\|V\|$	d_w	d_e	f_{e_b}	$d(f_p)$	f_t	$\|S_n\|$	L_w	f_m	ΔS_n
1	10000	100	$\{100, 50\}$	t	$\{64\}$	f_{t_1}	4	$\{1.0, 1.0\}$	f_{m_2}	0.2
2	10000	100	$\{50, 50\}$	t	$\{64\}$	f_{t_1}	4	$\{1.0, 1.0\}$	f_{m_2}	0.2
3	10000	100	$\{50\}$	t	$\{64\}$	f_{t_1}	4	$\{1.0, 1.0\}$	f_{m_2}	0.2
4	–	50	$\{50\}$	t	$\{64, 64\}$	f_{t_1}	2	$\{1.0, 1.0\}$	f_{m_1}	–
5	–	50	$\{50\}$	t	$\{64, 64\}$	f_{t_1}	1	$\{1.0, 1.0\}$	f_{m_1}	–
6	–	50	$\{100, 50\}$	t	$\{64\}$	f_{t_1}	4	$\{1.0, 1.0\}$	f_{m_2}	0.2

the model. `submissions_path` and `asts_path` are the paths to the submissions metadata and their encoded AST and all the data to train and evaluate the model will be loaded from their. The negative samples related settings as well as the class weights mostly affect the precision-recall trade-off, which we will discuss more in detail in 4.1.3.

4.1.3 Clone detection experiment

In this section, we will describe the experiments we run to evaluate our clone detection model. We will here describe the experiment we run, show the results and give an interpretation of the results we obtained.

First, we prepared the dataset to train our model by splitting the data we described in 4.1.1 into training set containing 80% of the data, the cross-validation set used to tune our hyper parameters containing 10% of the data, and finally the test set used to give a final evaluation of our model. For both training, cross-validation and test, we treat files implementing a solution to the same problem as clones, and randomly choose n samples from files implementing a solution to a different problem to use as negative inputs to our model. We make the number of samples vary during training but fix this number to 4 samples for cross-validation, in order to be able to compare the performance of the different models on the same input data.

We trained models to detect clone between Java and Python source code, as well as models to perform clone detection between source fragments written in Java. The main motivation for trying to run clone detection is Java is to be able to compare our results with other clone detection tools. In tables 4.6 and 4.8, we show the hyper parameters we used for training models for respectively Java/Python code clone detection and Java/Java code clone detection. Given the large number of parameters which we tried when training the models, we only show a subset including models which yielded the best results, as wee as models which we would like to discuss further in the results interpretation. Our models are sorted using the F1-score. In tables 4.7 and 4.9, we show different metrics for each model when tested on the cross-validation set.

In tables 4.6 and 4.8, having $-$ in the $|V|$ column means that we did not use the values of the tokens, and we did not have to limit the size of the vocabulary as it was already small enough. Concretely, the vocabulary size was of 130 for Python and of 75 for Java.

In general, our model gives us a high recall but does not manage to get a high precision,

Table 4.7. Java/Python clone detection models results

n	F1-score	Precision	Recall	Accuracy
1	0.66	0.55	0.83	0.83
2	0.63	0.51	0.80	0.80
3	0.56	0.42	0.82	0.73
4	0.55	0.40	0.84	0.71
5	0.52	0.38	0.84	0.68
6	0.49	0.35	0.80	0.66

Table 4.8. Java/Java clone detection models parameters

n	$\|V\|$	d_w	d_e	f_{e_b}	$d(f_p)$	f_t	$\|S_n\|$	L_w	f_m	ΔS_n
1	10000	100	$\{100, 50\}$	t	$\{64\}$	f_{t_1}	4	$\{1.0, 1.0\}$	f_{m_2}	0.2
2	10000	100	$\{50\}$	t	$\{64\}$	f_{t_1}	4	$\{1.0, 1.0\}$	f_{m_2}	0.2
3	10000	30	$\{50\}$	t	$\{64\}$	f_{t_1}	4	$\{1.5, 1.0\}$	f_{m_2}	0.2
4	10000	30	$\{50\}$	t	$\{64\}$	f_{t_1}	4	$\{2.0, 1.0\}$	f_{m_2}	0.2
5	10000	30	$\{50\}$	t	$\{64\}$	f_{t_1}	4	$\{1.0, 1.0\}$	f_{m_2}	0.2
6	–	50	$\{30\}$	t	$\{64\}$	f_{t_1}	1	$\{1.0, 1.0\}$	f_{m_2}	0.2

Table 4.9. Java/Java clone detection models results

n	F1-score	Precision	Recall	Accuracy
1	0.77	0.67	0.92	0.89
2	0.76	0.67	0.88	0.89
3	0.72	0.64	0.82	0.87
4	0.69	0.56	0.89	0.84
5	0.66	0.53	0.88	0.82
6	0.55	0.39	0.94	0.69

especially for the cross-language classification task. These results can be mitigated by trading recall for precision, but given the goal of our model to find possible duplicates across systems, we prefer to keep a high recall and leave the false-positive judgment to the developer. Furthermore, some source code which are actually not clones do look very similar, and can therefore be difficult to classify. For example, a code implementing Fibonacci and a code implementing factorial both contain a main for loop, which does some arithmetic operation and finally return an integer. Although the codes are not doing exactly the same thing, and are therefore not clones, they contain enough similarities to possibly be factorized, so the evaluation here could be a little pessimistic.

From the hyper-parameters point of view, the main conclusion to draw from the results we obtained is that when we do not include the values of the tokens and try to train our model with only the types, the resulting model does not contain enough information to be able to correctly detect clones and we have a large drop in the precision score. Another conclusion that we can draw from our results is that increasing the size of the LSTM encoder improves the performance of the model. Especially for cross-language detection, where the detection task is more difficult, we get a considerable improvement by using a stacked LSTM instead of a simple one. On the other hand, using algorithm f_{t_2}, which performs two depth-first search, for transforming our ASTs into vectors yielded performance much worse that the simple f_{t_1} which only perform a single one. We suspect that adding the result of the second depth-first search to the vector encoding added more noise to the encoded data that it added information, but a more formal conclusion about this result needs further investigation. We did not observe any significant improvement by changing the size of the output MLP classifier, and therefore concluded that the model performance was rather dependent on how we encode the AST into a vector rather the classification of the encoded vectors.

4.1.4 Comparison with existing methods

We chose to compare our method to SourcerCC[3], as it is the tool published the most recently with its source code publicly available. In the above experiment, we sampled 4 non-clones code pairs for each clone pair when evaluating our model. However, clone detection tools normally do not work by taking code pairs but search for clones in a set of code fragments. As we discuss in Section 5.2, our tool LearnerCC currently only works by taking code fragments pairs as input, and we therefore need to run $\mathcal{O}\left(n^2\right)$ comparisons to detect clones in the same way as the tool we are comparing our results to.

To run our experiment, we used the files from our test dataset, and run our tools using all the pairs combinations, as well as SourcererCC on the sampled files to detect clones within these files.

We will first describe the steps and settings we used to run the experiment using our model. To run the experiment, we select the model which performed the best in the above experiment according to the results presented in table 4.9. We generate all the possible code pairs of the files in our test set, and use our model to predict whether each of these pairs were clones or not. Using our predictions, which are probability-like values between 0 and 1, we set multiple thresholds to whether the clone pair is a code clone are not. If the threshold is closer to 0, we accept pairs that are less likely to be clones and therefore trade precision for recall, on the other hand if the

Table 4.10. Clone detection results

Threshold	F1-score	Precision	Recall
0.1	0.13	0.07	0.98
0.2	0.15	0.08	0.97
0.3	0.17	0.09	0.95
0.4	0.18	0.10	0.92
0.5	0.19	0.11	0.89
0.6	0.21	0.12	0.85
0.7	0.22	0.13	0.77
0.8	0.24	0.14	0.63
0.9	0.22	0.16	0.33

Table 4.11. SourcererCC settings

Parameter	Value
MIN_TOKENS	65
MAX_TOKENS	500000
BTSQ_THREADS	4
BTIIQ_THREADS	4
QLQ_THREADS	4
QBQ_THREADS	4
QCQ_THREADS	4
VCQ_THREADS	16
RCQ_THREADS	4

threshold is closer to 0.9, we are stricter for which pairs to accept and therefore trade recall for precision. We try a range of values for this threshold and compute the F1-score, precision and recall for each of them. We show our results in table 4.10.

After running the experiment with our model, we use the same dataset to run the experiment with SourcererCC. We show the parameters we used to run SourcererCC, which are the defaults at the time of the experiment, in table 4.11.

SourcererCC uses a threshold to decide whether it considers two code fragments to be clones or not. If the value of the threshold is 0.5, it means that at least 50% of the tokens must be shared between the code fragments to be considered as clones. The higher the threshold, the more the code fragments need to be similar to be considered as clones, therefore when the threshold becomes higher, the precision increases and the recalls drop. We show in table 4.12 the values we obtained for different values of this threshold.

In table 4.13, we choose the results for each tool which we think might be the most useful for

Table 4.12. SourcererCC clone detection results

Threshold	F1-score	Precision	Recall
0.3	0.05	0.03	0.29
0.4	0.08	0.05	0.24
0.5	0.12	0.10	0.15
0.6	0.11	0.24	0.07
0.7	0.05	0.63	0.03
0.8	0.02	0.79	0.01
0.9	0.01	1.00	0.00

Table 4.13. Clone detection results comparison

	F1-score	Precision	Recall
Our method	0.21	0.12	0.85
SourcererCC	0.05	0.63	0.03

the tool user — not necessarily the highest F1-score, but at least a good precision or recall — and compare them. In figure 4.2, we plot the precision-recall curve with the results of both our methods and SourcererCC. The x-axis is the recall and the y-axis is the precision, which means that we would like to maximize the area under the curve.

Our method and SourcererCC have very different characteristics and are therefore difficult to actually compare — our method has a very high recall but a poor precision, while SourcererCC has a good precision but a low recall. We will here try to give some explanation about the results we obtained. Our dataset is mostly composed of types IV clones, with probably only a very small amount of clones which would enter the type III clone category. SourcererCC is a token-based clone detection tool and is mostly design to detect type III clones, and given the nature of its algorithm should be particularly efficient for clones generated from copy-pasting. We believe that the poor results of SourcererCC on this dataset comes from its nature — clones in the dataset being introduced by different programmers implementing the same functionality, which generate very different types of clones. We think that the poor precision results with our approach comes from two different factors. The main important cause for this issue is that our tool is currently only able to predict if two code fragment are clones are not, and to be able to run this experiment, we therefore need to predict $\mathcal{O}\left(n^2\right)$ samples. The amount of clones in these samples is therefore less than 1%, which vastly lower than the 20% of clones we were using when training the model. This change in the training and test distributions therefore tends to wrongly predict two code pairs to be code clones. We describe more in detail in Section 5.2 a possible approach around this issue. Another reason for the precision to be low is the nature of clone detection itself. We currently mark two code fragments as clones only if they implement the

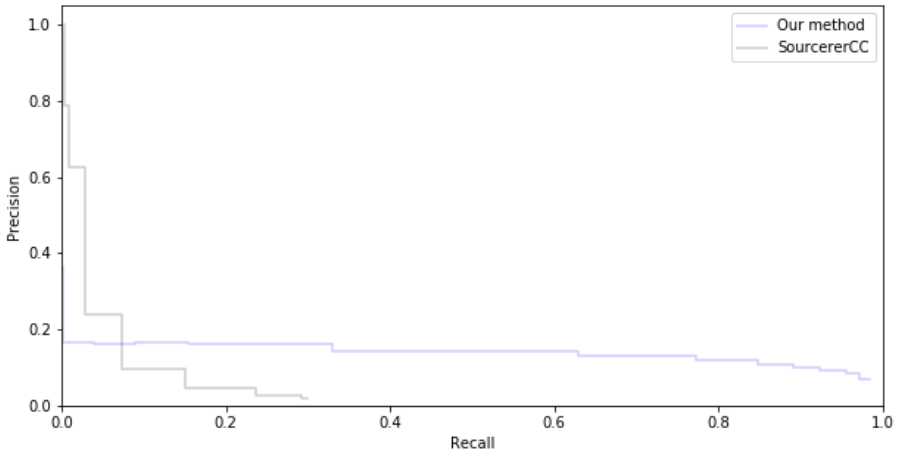

Figure 4.2. Precision-recall curve for clone detection experiment

exact same functionality. However, some code pairs can have a very close structure, which would make them good candidates for clones, but do not implement the same functionality. We show such an example in listings 4.3. Both code read an integer from the standard input, declare other integers, execute some simple arithmetic operations on the declared integer and finally output the the result. Both codes are therefore doing something quite similar, but do not enter in our current definition of code clones. However, the code fragments are similar enough to be predicted as clones by our model, which lowers the precision we obtain.

4.2 Token embedding generation

To generate token vector embedding and evaluate our results, we implemented the model and tools we described in Section 3.4. All the tools implemented to generated token embedding are publicly available on GitHub[1], and made available through pip when possible[2] [3].

4.2.1 Embedding generation dataset

As described in Section 3.4, data generation for token embedding is unsupervised, as opposed to the approach we take for clone detection. Getting data to train our model is therefore much easier. To train embedding for a particular programming language, the only thing we need is a large quantity of code written in this given language. This code should also be as much as possible diverse so we can capture the tokens of most major libraries and get a descent vector representation for their tokens.

[1] `https://github.com/tuvistavie/bigcode-tools`
[2] `https://pypi.python.org/pypi/bigcode-fetcher`
[3] `https://pypi.python.org/pypi/bigcode-embeddings`

```java
import java.util.Scanner;

public class Main{
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int ans = Integer.MAX_VALUE;
    for (int i = 1; i <= n; i++) {
      int j = n / i;
      ans = Math.min(ans, Math.abs(i - j) + n - i * j);

    }
    System.out.println(ans);
  }
}
```

```java
import java.util.*;

class Main {
  public static void main(String[] args){
    Scanner read = new Scanner(System.in);
    int x = Integer.parseInt(read.nextLine());
    int c = 0;
    int sum = 0;
    while (sum < x){
      c++;
      sum += c;
    }
    System.out.println(c);
  }
}
```

Listing 4.3. Structurally close code pair classified as clones

Table 4.14. Embedding generation Java dataset metrics

Metric	Value
Projects count	1,027
Files count	476,685
Lines count	80,367,840
Tokens count	301,930,231

Table 4.15. Embedding generation Python dataset metrics

Metric	Value
Projects count	879
Files count	131,506
Lines count	55,796,594
Tokens count	89,757,436

To generate datasets to train token embedding for Python and Java, we fetched code from GitHub using our `bigcode-embeddings` tool, and then generated the ASTs for all the files using `bigcode-astgen`. For Java, we chose to use all the projects written in Java and belonging to The Apache Software Foundation[4]. For Python, as we could not find any organization with a sufficient amount of source code, we projects which fulfilled the following conditions.

- Size between 100KB and 100MB
- Non-viral license (e.g. MIT, BSD)
- Not forks

We ordered the results by number of stars as a proxy of the popularity of the project and kept all the files which contained more than 10 tokens and less than 10000 tokens. Although our AST generation tool supports both Python 2 and Python 3, the produced AST may vary slightly and we therefore decided to use only Python 3 for this experiment.

We show some metrics for our Java dataset in table 4.14 and for our Python dataset in table 4.15.

4.2.2 Vocabulary generation results

As discussed in Section 3.4, the first step when creating token embedding is to generate a vocabulary for the given programming language. We will here present the results for the different

[4] `http://www.apache.org/`

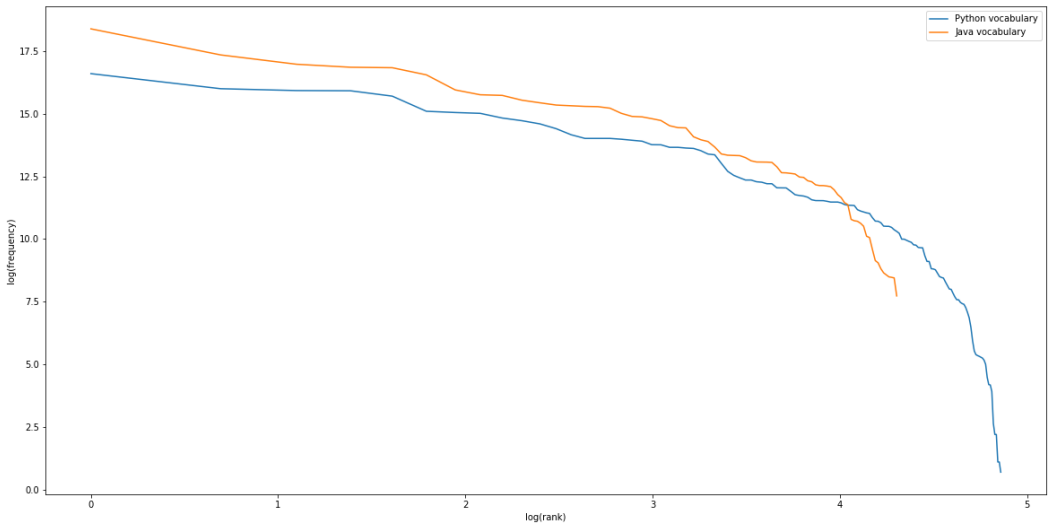

Figure 4.3. Tokens without value relation between token frequency and rank

vocabularies we generated. For both Java and Python, we generated 3 different kinds of vocabularies:

1. Vocabulary without token values
2. Vocabulary of 500 tokens with values
3. Vocabulary of 10000 tokens with values

The vocabulary without token values only contains only the type of each token, for example, `ImportStmt` in Java, or `FunctionDef` in Python. In figures 4.3 to 4.4, we show the relation between the rank of a token in the vocabulary, and its number of appearances. We use a log scale for both axes. An interesting property we find is that, when using values of the tokens, this relation and by extension the distribution of the frequency tokens in a programming language follows the Zipf's law [20], in the same way as natural languages do. When we exclude the tokens, and therefore only use the type of each token to generate the vocabulary, the law seems to hold only until the few last tokens of each language, which indicates that there is a small number of types in each language that are almost never used. Regardless of the fact that we use or not values of each token, the distribution is almost the same for both Python and Java, and the fact that the curve for Java is above the one for Python comes from the fact that we extracted more tokens for Java and we did not normalize the results, as it was not needed for the rest of the embedding generation.

Given the distribution of the token frequency in both programming languages follow closely the frequency found in natural languages, we can expect that a model such as skipgram which works well with natural languages to also work here.

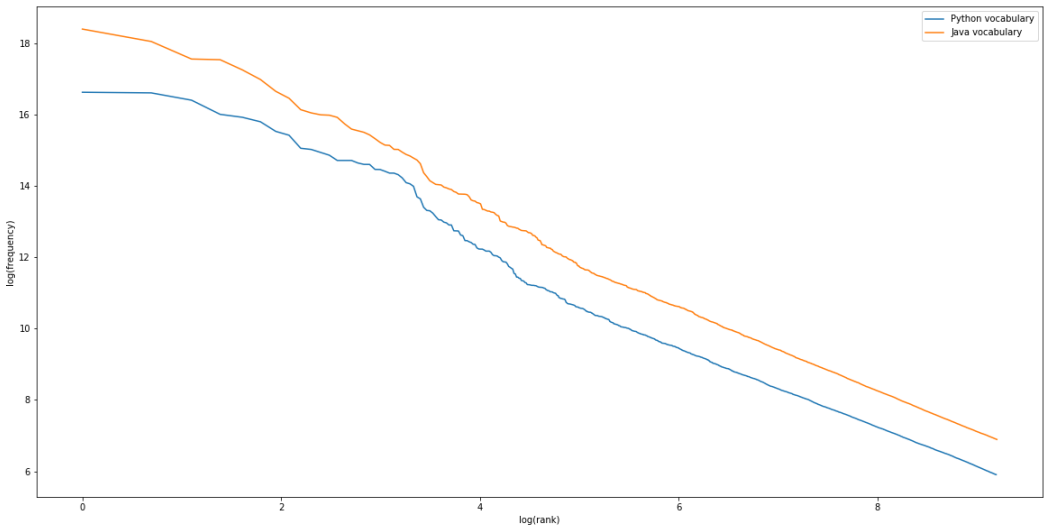

Figure 4.4. Tokens with value relation between token frequency and rank

Table 4.16. Experiment settings for embedding generation

Parameter	Description	Sample value
`vocabulary`	The vocabulary to use for generating the embeddings	`no-id.tsv`
`ancestors_window_size`	The window size for the number of ancestors	2
`descendants_window_size`	The window size for the number of descendants	1
`include_siblings`	Whether or not to include siblings	**`true`**
`embeddings_size`	The dimension d_w of the embeddings	100

4.2.3 Embedding generation experiments

Once we created our vocabulary, we want to assign a vector in $\mathbb{R}^{d_w}$ to each token in our vocabulary. Given the purpose of generating these embedding is to encode a program so that we can detect clone, the main requirement of our embedding is that tokens which are semantically similar, such that for example **`while`** and **`for`** have a small distance, while unrelated tokens have a large distance.

In table 4.16, we describe the different settings of our experiment. We only show the most important for the task of generate the embedding and leave out settings such as the batch size, the optimizer or the learning rate.

Given the unsupervised nature of the model, it is difficult to evaluate quantitatively the quality of the results. While some methods exist to evaluate word embedding in natural languages context [21], the technique uses the nature of the language and is therefore not directly applicable to programming languages. We are not aware of such a method to evaluate embedding in

Table 4.17. Embedding experiment optimal settings

Parameter	Vocab without values	Vocab with values
`ancestors_window_size`	2	2
`descendants_window_size`	1	1
`include_siblings`	**`false`**	**`false`**
`embeddings_size`	50	100

programming languages context. We therefore decided to evaluate quantitatively the results we obtained for the embedding creation task by manually inspecting the relation between different tokens, and judging of the quality of the embedding using these relationship. In particular, when training embedding for vocabulary where token do not have values, we tried to cluster the tokens using k-means[22] and looked at how much the clusters made sense — for example, are statements and expressions in different clusters.

We tried to train the embedding using a large set of values for the different parameters we had. We tried window sizes from 0 to 5 for the ancestors, from 0 to 4 for the descendants, we tried to use siblings and we tried embedding sizes of 10, 20, 50, 100 and 200. When increasing the size of the embedding did not present any significant benefit, we kept the smaller size. In table 4.17, we show the parameters we found to work best for training embedding for vocabulary containing values and vocabulary with types only.

Increasing the window size too much seems to create too much noise, and did not yield better results. Likewise, we suspect that including the siblings generated more noise than signal when training our model. In figure 4.6 and 4.5, we show the results we obtained when projecting the trained embeddings in a 2 dimensions space using PCA[23]. We only plotted a subset of the points in the vocabulary to keep the figure readable.

There are a few interesting things to notice in the results we obtained in figures 4.6 and 4.5. First, the clustering we obtain seems quite close to what we could expect when thinking about the different types of tokens present in the AST. For example, for Java, All the statements are part of the same cluster, and the only expression which belongs to this cluster is the `LambdaExpr`, which was introduced in recent versions of Java and appears way less often than most other tokens, which could explain why it was not clustered correctly. The expressions are also mostly in the same cluster, and literal expressions, such as `IntegerLiteralExpr` and `StringLiteralExpr` are close to each other, while other related tokens such as `ThisExpr` and `SuperExpr` are in the same cluster but further away in the vector space. The results are similar for Python, where most language constructs, such as `While` or `Try` are clustered together, while all the comparison operators are in another cluster. Another interesting point is the difference in the relationship of the tokens between both programming languages. In particular, while `While` and `For` appear almost at the same point in the vector space for Java, which means that both tokens are very similar, the same two tokens are much further away in Python. This seems a reasonable result to obtain as Java's **`while`** and **`for`** are indeed interchangeable, while in Python, these two constructs have different semantics.

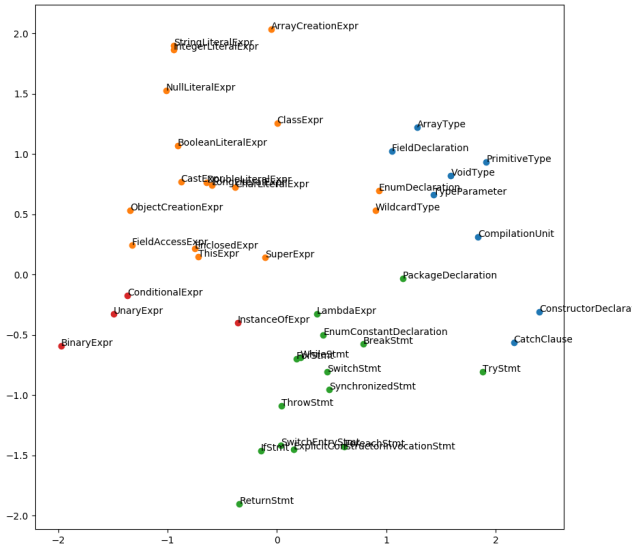

Figure 4.5. Java clustered embedding 2D projection for vocabulary without values

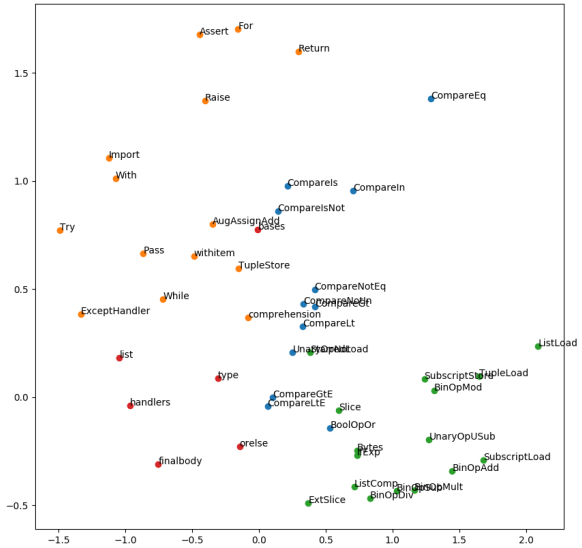

Figure 4.6. Python clustered embedding 2D projection for vocabulary without values

Chapter 5

Conclusion

5.1 Summary

In this thesis, we presented a system capable to detect code clones across programs written in different programming languages.

We gave some background about the motivations behind cross-language clone detection, and explained the difficulties that can be encountered when trying to achieve this task. In particular, we gave some explanation about why many approaches used for clone detection in a single programming language cannot be applied directly, and why machine learning seems to be a good fit for this task.

We then presented a supervised learning based machine approach to detect code clones. We introduced an LSTM based model which predicts if two programs are clones or not, we gave details about how we compute token embedding for a particular programming language, and showed how we can pre-process source code and use these embedding to feed data to our model.

We introduced a novel dataset for cross-language code clone detection composed of programs written in Java and Python, containing around 50000 files and implementing more than 500 different functionalities. We explained the motivation behind the choice of competitive programming to collect programs for this dataset, and gave details about the methodology used to generate the dataset.

We then showed the results we obtained when evaluating our trained system using our dataset. We demonstrated that we are able to obtain a high recall, with scores around 0.7, while keeping a reasonable precision — with scores in the order of 0.3 to 0.5 — given the motivation of our system. We also presented the results we obtained when training token-level embedding using a large set of data we fetched from GitHub, and described how it helps us achieve better results for our clone detection task.

5.2 Future work

Although we obtained some promising results for the task of cross-language code clone detection in terms of recall and precision, scalability is a major issue of our current approach. In the current implementation of our system, our model takes as input two code fragments, and determines if

they are code clones or not. This works correctly when evaluating our model by feeding it pairs of clones directly. However, in a real-world use case, this means that the clone detection process would take $\mathcal{O}\left(n^2\right)$, where n is the number of code fragments to evaluate. As code fragments are most commonly code blocks, and the number can therefore be in the order of 100 millions in a large enough project, our method as is would not work. We should be able to modify our system to add a hash-layer as done for example in [12] and use a method such as locality-sensitive hashing [24] to reduce the order of our system to $\mathcal{O}(n)$, making it suitable for real-world applications.

References

[1] Analyzing github, how developers change programming languages over time. `https://blog.sourced.tech/post/language_migrations/`.

[2] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, 115, 2007.

[3] 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*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.

[4] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 1992.

[5] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[7] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[9] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[10] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

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

[12] Ming Li Huihui Wei. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 3034–3040, 2017.

[13] Nghi D. Q. Bui, Lingixao Jiang, and Yijun Yu. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. *CoRR*, abs/1710.06159, 2017.

[14] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM.

[15] Nicholas A Kraft, Brandon W Bonds, and Randy K Smith. Cross-language clone detection. In *SEKE*, pages 54–59, 2008.

[16] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[18] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 761–774, New York, NY, USA, 2016. ACM.

[19] Viacheslav Khomenko, Oleg Shyshkov, Olga Radyvonenko, and Kostiantyn Bokhan. Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization. *CoRR*, abs/1708.05604, 2017.

[20] Steven T. Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic Bulletin & Review*, 21:1112–1130, 2014.

[21] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 298–307, 2015.

[22] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, July 2002.

[23] Jonathon Shlens. A tutorial on principal component analysis. *CoRR*, abs/1404.1100, 2014.

[24] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

Acknowledgements

I would first like to thank my supervisor, Professor Shigeru Chiba, for all the guidance he provided me with during the course of my master's degree. He has always been very available and helped me through many instructive discussions, not only to improve and consolidate my research, but also to gain more insight about research and life in general.

I also want to thank all my colleagues of the Core Software Group for all the insightful conversations and of course for the nice two years we spent (and coffee we had) together.

Finally, I would like to express my sincere gratitude to my parents who have supported me during all the course of my studies and without whom I would not have had the opportunity to follow this degree.