Code Preparation for Machine Learning

Converting code into vectors or sequences

Raphael Jenni
OST Eastern Switzerland University of Applied Sciences
MSE Seminar "Programming Languages"
Supervisor: Farhad Mehta
Semester: Spring 2022

Abstract

Using machine learning for images, text, or audio has become popular and relatively mainstream. On the other hand, using machine learning for code is a rather new field. Only a few commercial products are available, and the research is still in its early stages. When trying to join this field, many different topics need to be explored.

This paper aims to bring a software engineer or a programming language researcher up to speed on the current state of machine learning and show the possibilities of such technologies in respect to code. It covers code2vec, code2seq, CuBERT, CoCluBERT, CodeBERT, TreeBERT, and DeepBugs AST Context Representations, with their respective backgrounds and list tools and points out available further research. It also covers a few use cases and gives a practical example that leads through the whole paper.

Keywords: Programming Languages, Pre-processing, Deep Learning, Code2vec, Code2seq

1 Introduction

Applying Machine Learning (ML) for Images, Text, or Audio has become normal. The available tools and the possibilities for such use cases are plenty and very impressive. Not the same can be said for applying ML when working with code. In recent years a few products emerged like the GitHub Copilot[1], or TabNine[2], both tools for code completion, or some research projects like the OpenAI Codex[3] have been released. All of them are very interesting and sometimes even useful, but within a limited scope and for just a particular field. One primary reason for that is that there is a lack of standards or well-proven ways to handle code for machine learning applications. Another reason is that gathering good code samples is an arduous task. This paper details the current state of representing code to learn from it and its pitfalls.

1.1 Overview

In order to explain the way code can be represented, some basic concepts are needed. We start with an example, set the goal we want to achieve, and gradually go into the underlying ideas. The example is taken from the paper code2vec [AZLY19], more on that later. The goal is to predict the function name of a given function automatically.

For example, we want to input the following code snippet (Listing 1) and want a model to predict that this functions name is `isPrime`.

```java
boolean f(int n) {
  if (n <= 1) {
    return false;
  }
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      return false;
    }
  }
  return true;
}
```

Listing 1. isPrime Code Snipped

In order to achieve that, several problems come up:

First, there is quite a lot going on in the code. It takes an understanding of the programming language, an understanding of what a prime number is, and how it can be calculated. It also requires the knowledge of natural language to give that function a meaningful name. All requirements that are pretty simple for a human, or at least for a reader of such a paper, but not for a machine. This is what section 2 will be about: *Why is it hard for a computer to learn from code, how is code different from natural language, and how does a computer learn in general?*

Second, how can text be handled? Text is not just a string of characters or a sequence of words but has some underlying structure. Nevertheless, some knowledge gained from natural language processing (NLP) can be transferred when working with code. More details on this topic is covered in section 3: *What do we understand by NLP, what are word embeddings (word2vec), what are sequence to sequence models (seq2seq), what is BERT, the model used in Google Search, and how can we use them to learn from text?*

Third, how can the computer take the code and make sense of it? There are several methods to represent code, but they differ in the amount of information they provide, the speed of computing them, and how well they are suited for specific tasks. What are the tradeoffs between them, and what is the right one for our task? In section 4 we try to

[1] https://copilot.github.com/
[2] https://www.tabnine.com/
[3] https://openai.com/blog/openai-codex/

pass on the knowledge to let you answer those questions for yourself.

The focus will lie on code2vec and code2seq, as their approach is simple to understand, well generalizable, yet powerful. In section 5 their details are described together with the analysis of our `isPrime` example.

If the paper has stirred your interest in this field, and you want to find out more, or try it out for yourself, section 6 lists some tools, already implemented use cases, and further research that has been conducted.

1.2 Assumed Knowledge

This paper assumes that the reader knows the basics of programming language constructs and generally understands machine learning.

2 Problem

As mentioned previously, it is hard for a computer to interpret what a code snipped does. One may ask why, as understanding natural language is already possible and seems much more complex than understanding code, with which the computer already works with it. This question is valid, however the issue is that the computer just runs code and does not need to understand it. Code is to a computer what a recipe or manual is to a human. It just describes *what* needs to be done and *how* it needs to be done but does not state what the whole program does. This dilemma is also present for code verification or the famous halting problem [Tur37]. How can we verify that the code is doing what it is supposed to do, or how can we be sure that it does not get stuck in an endless loop without executing it? One can do that for simple programs, but as soon as the code gets more complicated or includes some recursion, it is impossible. If there is an input involved, which is the case most of the time, one would have to run the function with every possible input. The possibilities of a function's outcome are infinite.

Let's go back to the reference to natural language. Google, for example, seems to understand what people are searching for. Therefore, it needs to understand the written text. This assumption is valid, but behind its intelligence lies a massive neural network trained on hundreds of millions of questions and answers until it learned to "understand" the written text. The word "understand" is written in quotes because neural networks represent everything internally as numbers. Each word or word combination is represented by a number. Each connection is represented by a different number, and the result is again a vector of numbers that just gets translated back into a series of words. In order to understand code in the same way, it also needs to be converted into numbers.

So we can think of code as being similar to natural language but with a much better structure. This structure is something we can and should take advantage of. The question is just how. More on that later in section 4. Furthermore, lots of training data is needed. Luckily, as open-source software has become more common, a lot of code is available for free. The problem comes from the fact that the quality of that code is unknown, and for many tasks, no accurate labels are available.

3 Background in Natural Language Processing (NLP)

Going back to Natural Language Processing *(NLP)*, it is essential to understand some basic functionalities. NLP is all about learning and understanding the meaning of words and phrases. A text consists of phrases, and a phrase consists of words. In order for a computer to understand that, we first need to convert the input into a sequence of tokens that can then be converted into numbers. This is called *tokenization*. A token is a predefined subsequence of the input. In its lowest form, a token could represent a single letter. Therefore, a word would be a sequence of such tokens. Taking each letter as a token would undoubtedly be possible but would likely not yield any beneficial results. The granularity would be just too high. Usually, a token represents a word or a so-called *n-gram*. An n-gram is a sequence of n words. For example, the sentence `I like dogs` would be represented by the three word tokens `I`, `like`, and `dogs` or the 2-grams `I like` and `like dogs`. Each token gets assigned a number, often according to the order of appearance in the text or the number of occurrences. The final output of the tokenizing process is then the sequence of numbers representing the input. With this tokenized sequence, the computer can start learning. There are several advanced text pre-processing and tokenization methods, but those are not relevant for our use case.

3.1 Word2Vec

Tokenizing a sequence is often not enough to learn from the input. The numbers are just identifiers. They do not have any real connection to one another other than the natural order of numbers. But even this order does not get used, as it has no meaning in regards to words or sentences in the context of natural language. We could convert the sequences to a so-called *one-hot-encoding*, which is a sequence of binary numbers. For each type of token, a column is assigned. The number `1` being the first column, `2` the second and so on. Each row then consists of zeroes except for the column of the token. The one-hot-encoding is a straightforward encoding that shows excellent efficiency for a small number of tokens. However, it becomes much less efficient when the number of tokens or, more generally "categories" increases, as it is very sparse[4] and "wastes" a lot of memory. A typical maximum number of categories is `50`. With this limit, using the one-hot-encoding is not suited for encoding natural language as the number of tokens is well beyond 50 tokens.

[4]Sparse: Meaning containing a lot of zero values.

This is where *embeddings* come in. Embeddings are a way to represent a sequence of numbers through learning the connection between the tokens in a much smaller space. For example, words such as “France”, “Spain”, and “Germany” should be clustered closer together, than “France”, “Tree”, and “Computer”. Embeddings are represented by an embedding matrix of size *vocabulary size* $\times$ *embedding dimension*, the embedding dimension usually ranging between 10 and 300. This matrix is initialized randomly and slowly adjusts during learning to represent the categories in the vector space. This process is visualized in Figure 1. The interesting feature of

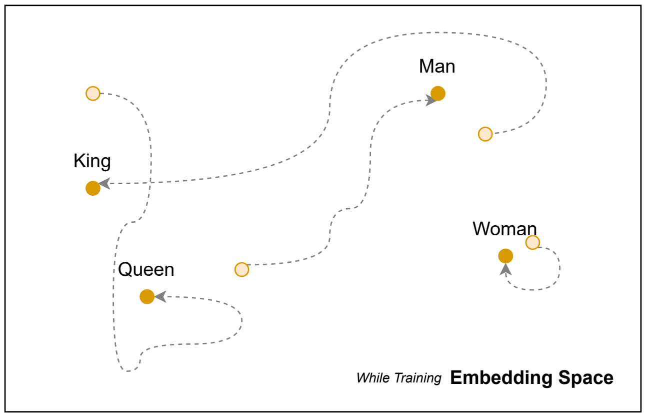


Figure 1. Visualization of the Initialization and Training of the Embedding of “King”, “Queen”, “Man”, and “Woman”.

the closeness is that similar representations are clustered closer to each other, and vector operations can be performed on them. A famous example is called the “King - Man + Woman = Queen” example. It must be noted that embeddings need to be learned as part of the actual training process or imported as already pre-trained. For example, in the case of the “King - Man + Woman = Queen” example, the actual training process could be assigning book titles to several genres. The resulting embedding is just a nice by-product that can later be used for other tasks in the same domain. Although it is not uncommon that there is only interest in this by-product.

By adding and subtracting the embedding vectors of the words, the resulting vector will be very close to the word “Queen”. This is visualized in Figure 2. A similar example would be “Madrid - Spain + France = Paris”. This type of converting words into vectors is called *Word2Vec* and are widely used. [Gér19, Mig17] As embedding data is a very common task in machine learning, all the major machine

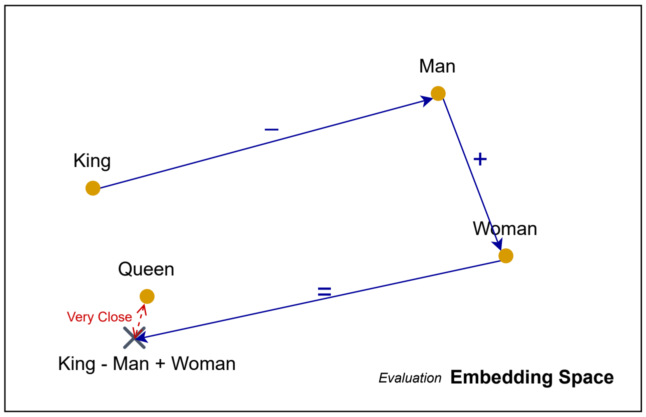


Figure 2. Visualization of the Evaluation of the “King - Man + Woman = Queen” Example.

learning frameworks include this functionality out of the box [5,6,7,8,9].

3.2 Seq2seq

Another machine learning model type is the sequence to sequence model, short seq2seq. Seq2seq models are used to convert one sequence to another, as the name suggests. In natural language processing, two primary use cases are translating sentences from one language to another and “answering” questions as, for example, Google Search does it. Seq2seq models always work in two parts. First, we have an encoder. An encoder is a network that takes a sequence of input tokens and produces a sequence of hidden states. The hidden states’ sequence does not have to be the same length as the input. It can even consist of only a single number. This hidden state then gets passed to the decoder, which takes the hidden state and produces another sequence of output tokens. This is often also called an encoder-decoder architecture. In the example of language translation (see Figure 3), the input would be a sentence in one language, and the output would be the translation of that sentence in another language. However, seq2seq models are not limited to natural language processing; they can be used for any sequence to sequence conversions, like for video, audio or electrical signals. [Ala, Goo]

3.2.1 BERT. A model created by Google in 2018 is called BERT [DCL+]. *BERT* stands for "Bidirectional Encoder Representations from Transformers". The model is designed to pre-train text representation that can easily be adjusted for various tasks. One significant change that differentiates it

[5]https://keras.io/api/layers/core_layers/embedding/
[6]https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding
[7]https://scikit-network.readthedocs.io/en/latest/reference/embedding.html
[8]https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
[9]https://spark.apache.org/docs/2.2.0/mllib-feature-extraction.html

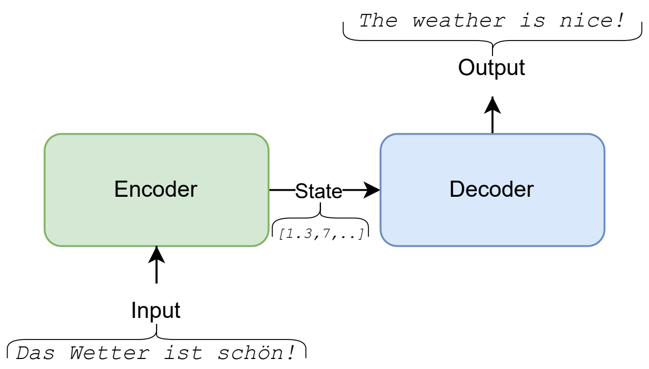


Figure 3. Simplified seq2seq model for translating sentences from German to English

from its predecessors is that the model reads text not just from left to right but simultaneously also from right to left. This is called *bi-directional* and can leverage the context on both sides of a word. This type of reading is essential because BERT was trained to gather an in-depth understanding of the language it was supposed to learn. During training, 15% of the tokens were masked, and it was BERT's job to predict the correct token. As this requires a deep understanding of language, and not everything can be inferred from the start of a sentence, this bi-directional property was a significant leap forward. A second task BERT was trained on is to predict the following sentence when receiving an initial one, called Next Sentence Prediction (*NSP*). This model was a breakthrough in machine learning for NLP and is, together with its descendants, now used in a wide range of applications. Google internally started to use BERT in its search engine in 2019.

4 Understanding Code

Returning from the digression into NLP, we can now start to look at the specifics of working with code in the context of machine learning. This section will cover different possibilities we have to work with code and explain how to interpret and understand it.

4.1 Code as Natural Language

The simplest form to understand code is to look at it as natural language. For example, the code snipped `if (x > 0){y = 1;}` can be tokenized as `if, (, x, >, 0, ), {, y, =, 1, ;, }`. For those tokens, we can then create an embedding and try to learn the meaning of the code. This requires a lot of data to gain real insights, which can be very computer resource-heavy, time-consuming, and therefore costly. Nonetheless, two similar models apply the BERT logic for programming languages.

4.1.1 CuBERT.

Code Understanding BERT, or short *CuBERT* [KMBS20], is a model for code understanding based on the previously covered BERT. It is built the same way as the BERT model but with code instead of text. The model is trained on 7.4 million python files and does not differentiate the code and some text that may be in the code in the form of comments. It aims to create a contextual embedding of source code. To improve the model's performance, six additional tasks were performed:

- **Variable-Misuse Classification** tackles the use of an incorrect variable in the place of a correct variable, that may occur by carelessly copy-pasting code and forgetting to rename variable occurrences. The task is to predict whether there is a variable misuse in the function. This task was proposed by Vasic et al. [VKM+] based on the slightly simpler task by Allamanis et al. [ABK].
- The **Wrong Binary Operation**-task is a task by Pradel et al. [PSD17, PDK18] for detecting whether a binary operator in a given expression is correct. An example would be using `i <= length` instead of `i < length` in a condition. Negative training examples are automatically generated by replacing some binary operators with another compatible operator.
- The **Swapped Operand** task handles cases where operands of non-commutative binary operators are swapped. An example would be a case where `list.size()< x` is wrongly expressed as `list.size()> x`.
- The **Function-Docstring Mismatch**-task handles cases where the docstring attached to a function corresponds to an other function. Something that can easily happen when carelessly copy-pasting functions.
- The **Exception Type**-task is a classification task where a missing exception token is predicted. The goal is to prevent users from using generic, catch-all, exception handlers.
- The **Variable-Misuse Localization and Repair**-task is based on the variable misuse task mentioned before. The goal is to predict the location of a misused variable (localization) together with the correct variable that should have been used (repair). It also covers the case where no bug is present in the code, which gets reported as such.

In all the tasks, CuBERT outperformed previous methods by a margin of 3.2% to 14.7%.

4.1.2 CoCluBERT.

Based on CuBERT, Code Clustering BERT, or short CoCluBERT [HPP+] was created to cluster source code. The goal is to group and categorize source code that is not labeled by its functionality. This should enable ML engineers to understand code better and provide the possibility to easier find and manage relevant pieces of code. Three different variants of CoCluBERT were the result of the paper [HPP+]. They were able to perform unsupervised machine learning to group source code by functionality in well-separated and compact clusters.

4.1.3 CodeBERT.

Another model called CodeBERT [FGT+], similarly to CuBERT, is a model based on the BERT architecture for programming languages. It supports six programming languages and is different from previous works by not using only *bimodal* data, but also larger amounts of *unimodal* data. *Bimodal* data is data that comes in pairs of two different types, such as pairs of language-image, language-video, or in this case natural language-programming language (*NL-PL*) pairs. *Unimodal* data on the other hand is data that is standalone, in this case code without any paired documentation. The model was trained for two objectives: Masked Language Modeling (*MLM*) and Replaced Token Detection (*RTD*). MLM's objective is to predict the token that was *masked* in the input sequence. RTD's objective is to predict the token that was *replaced* in the input sequence. The final objective of CodeBERT is to provide a pre-trained model for applications such as natural language code search or code documentation generation.

Both CuBERT and CodeBERT are similar in the way they apply the BERT approach to code understanding. CuBERT is the first model that attempts to pre-train contextual embeddings for code, and CodeBERT is the first uni- and bimodal pre-trained model. It will be interesting to see how these two models compare on different tasks. Unfortunately, only limited comparisons are available when writing this, and no clear advantages of one over the other are evident.

4.2 Leverage Code Structure

Although treating code as if it were natural language works in the setting for searching for code and connecting it with natural language, the fundamental structure that code is built upon is neglected. Code has a very well-defined structure, and this structure is one of the significant benefits code has over natural language when it comes to extracting information out of it. A programming language is formally specified. The grammar clearly and completely defines what the code has to look like and how all the tokens relate to each other. For example the grammar[10] (example in Listing 2) or the syntax definition[11] (example in Listing 3) for Java is available on the internet.

```
IfThenStatement:
    if ( Expression ) Statement
```

Listing 2. Example Java Grammar

```
Statement:
    ...
    if ParExpression Statement [else Statement]
    ...
```

Listing 3. Example Java Syntax

[10] Java Grammar: https://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html

[11] Java Syntax: https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html

In the two examples, a `ParExpression` is the same as `( Expression )` and clearly states that parentheses are expected to surround the expression. Every statement is required to have an expression and at least one statement. Not trying to leverage all this information would be a missed out chance. This brings us to the next possibility for understanding code.

4.3 Code as a Graph

Code can be represented as graphs in many ways. One can use static and dynamic techniques to analyze code automatically and gather such graphs.

Static techniques examine an application's source code without running it. A *control-flow graph (CFG)*, for example, represents all paths a program can be traversed through during its execution. This graph can contain loops and grow very large, depending on the analyzed program. Based on this, *data-flow graphs*, are graphs that show the flow of data through the code. The vertices represent all the places where variables get assigned or used. The edges represent the relationships between the usages of these variables. Those graphs can help to get a great understanding of the code and its internal workings.

Dynamic techniques, on the other hand, examine an application during runtime. To gather dynamic information, one would need to execute code. Execution graphs, for example, are condensed CFGs that describe the execution of the code and show what parts have been executed and what parts have not. Executing code is often not possible and, most of the time, not wanted, as it can take a long time and increases the complexity.

Code representations used for machine learning, therefore, use static analysis. The mentioned graphs contain much information and would be extremely valuable for learning. Representing a graph as a matrix would also be straightforward. The easiest way being an adjacency matrix. The big problem they bring with them is that in order to construct the graph, a lot of information needs to be computed. Doing that for the scale needed for applying machine learning afterward is often not feasible and would be very time-consuming. The time one often does not have. Therefore, those meta representations are instrumental in traditional code analysis but are not the best suited for machine learning. A more general kind of graph would be the abstract syntax tree.

4.4 Abstract Syntax Tree (AST)

The Abstract Syntax Tree (*AST*) is, as the name might suggest, a tree representation of the code. For example, the code snipped `if (x > 0){y = 1;}` can be represented as AST as shown in Figure 4. To create an AST, we need to know the grammar and parse the written text according to it. Provided, the written code is syntactically correct, the AST can easily be created. Having all this structure and information allows us to get an exact image of all the operations present in the code and their order. The problem with this representation is

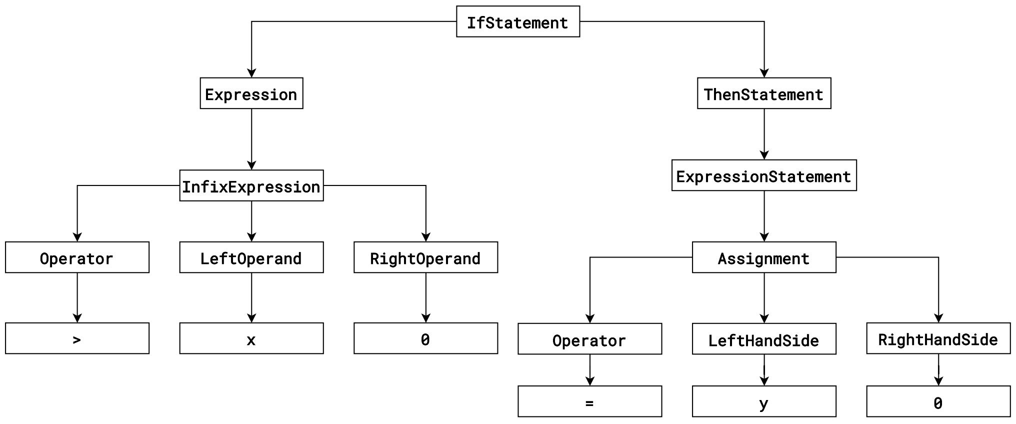


Figure 4. AST Representation of `if (x > 0){y = 1;}`

that it is not very easy convertible into a format that can be used for machine learning. Keeping too much information, the resulting vector will be vast and difficult to handle efficiently. It is a tradeoff between keeping as much information as possible and simultaneously creating the smallest possible vector that a machine learning model can interpret.

4.4.1 DeepBugs AST Context Representation.

The paper “DeepBugs” [PSD17, PDK18] created a representation where a context, consisting of the parent node, position in the parent, siblings, uncles, etc., for each node is created. Each node in the AST has an index by which it is referenced. This context is then one-hot encoded. This approach works for the purpose of finding simple bugs like, for example, the previously mentioned "Wrong Binary Operation" task. It is currently unknown if this approach would work for other use cases as well. For an example, checkout Michael Pradel’s DeepBugs repository on GitHub[12] or have a look at the paper "Machine Learning for Programming Languages" [Jen21].

As such, an AST can get rather large. There are many columns in the vector, and therefore the resulting vector is very sparse. The resulting vector’s size and the representation’s complexity may not be optimal for general use cases. Handling all this data would result in an enormous model size and memory footprint. Nevertheless, the DeepBugs project is open source and still actively maintained and improved when writing this.

4.4.2 Paths in the AST.

A more straightforward solution is to create a representation of all paths from the root down to the leaves in the AST. Looking at Figure 4 again, such a path could go from the root `IfStatement` to the right-most `0`. This would look something like: `IfStatement, ThenStatement, ExpressionStatement, Assignment, RightHandSide, 0`, also visualized in Figure 5. This path would then need to be tokenized into a vector in order for it to be used in a machine learning task. Such a representation would contain all the information and

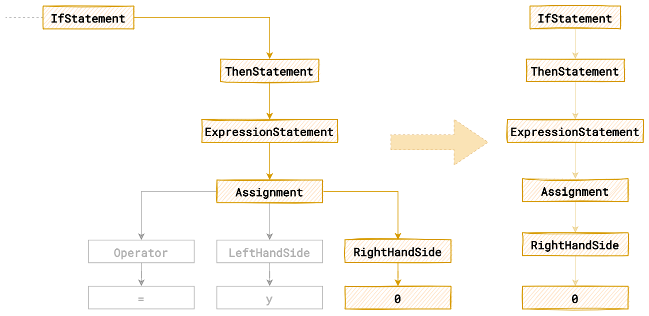


Figure 5. AST to Path Example (Root to Leaf)

could also be reversed back to the original AST, provided the order is kept consistent. It is also effortless and fast to compute. If nodes of the same type are represented by the same number in the tokenization process, the resulting vector could also be relatively small and, therefore, easy to handle. By doing this, the resulting size is reduced, but any AST position agnostic feature of the node is also removed. No matter what we do, the primary issue with this representation is that there is not much context information in a single path. Only combined with other paths or when including unique identifiers of the nodes context information can be extracted. The following model uses an AST representation based on this.

4.4.3 TreeBERT.

Another BERT-based model is TreeBERT [JZL+]. Both, CuBERT and CodeBERT have noted that using the syntax tree to improve the understanding of code is a possible improvement the authors are interested in looking into. This is precisely what TreeBERT did. TreeBERT represents the AST as a set of paths and adds a node position embedding. Just like CodeBERT, TreeBERT uses masking to learn to understand the AST and infer missing parts of the AST. It first converts the AST into a set of paths, with some nodes being masked. It then transforms the paths into a vector

[12] https://github.com/michaelpradel/DeepBugs

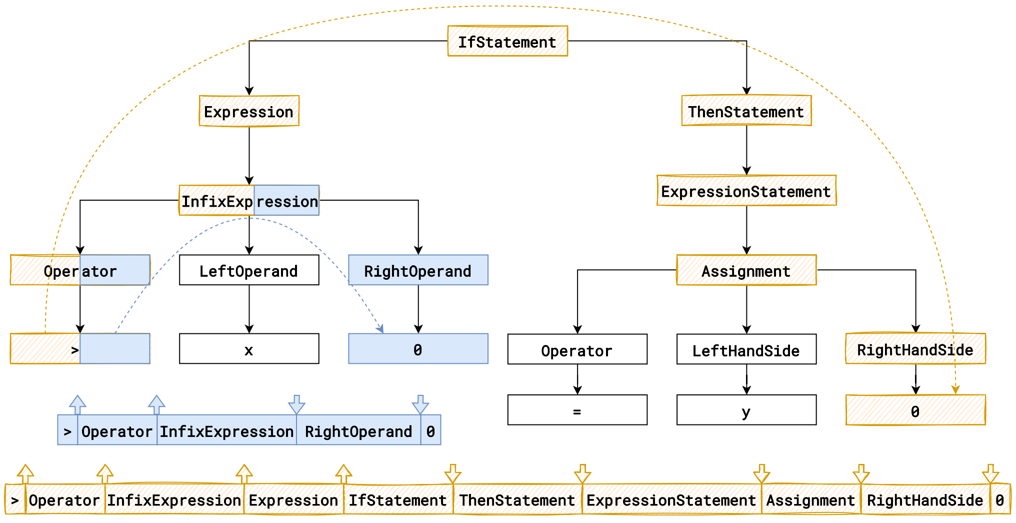

IfStatement
Expression
ThenStatement
ExpressionStatement
InfixExpression
Operator
LeftOperand
RightOperand
Assignment
>
x
0
Operator
LeftHandSide
RightHandSide
=
y
0
> Operator InfixExpression RightOperand 0
> Operator InfixExpression Expression IfStatement ThenStatement ExpressionStatement Assignment RightHandSide 0

Figure 6. AST to Path Example (Leaf to Leaf)

representation and predicts the correct AST. This prediction consists of inserting the missing/masked nodes, called "Tree Masked Language Modeling" (*TMLM*), and putting the nodes in the correct order, called "Node Order Prediction" (*NOP*). The results presented in the paper for code summarization and code documentation give TreeBERT the edge over CuBERT, CodeBERT, and Code2Seq. The author plans to improve the model and use TreeBERT for more and different programming language-related tasks.

4.4.4 Relationship Paths in the AST.

A similar approach to creating the paths from the root to the leaf, proposed by Elon et al. in "A General Path-Based Representation for Predicting Program Properties" [AZLY] and later used in "Code2Vec" [AZLY19], is to create paths from one leaf to another. For the example of Figure 4, the path from the left most leaf ">" to the right most lead "0" would be: `>, Operator, InfixExpression, Expression, IfStatement, ThenStatement, ExpressionStatement, Assignment, RightHandSide, 1`. However, such a path could also only go from the leaf ">" to the leaf "0" in the same subtree: `>, Operator, InfixExpression, RightOperand, 0`. Both paths are visualized in Figure 6. As a side note; in the original paper, each node in the path was also decorated with the direction of the tree traversal, meaning "UP" or "DOWN" for going up or down the tree. In the second paper, "Code2Seq" [ABLY19], this information was dropped. If the paths are created for every leaf, all the relationships between them are covered. As this may lead to an enormous number of paths, it is advisable to only select paths that have a limited length, meaning not going too far up the AST. This, in turn, may reduce the amount of captured information. We will now investigate this approach more thoroughly.

5 Code2vec and Code2seq

We want to focus on the code2vec and code2seq approaches, as they are pretty simple to understand yet powerful and with many possibilities. Furthermore, it does not require any special pre-trained model and can be easily adapted to any programming language. Let us start with code2vec, which came first and builds the basis for the code2seq approach.

5.1 code2vec

Code2vec works by taking a subset of code that is syntactically correct and can be parsed into a complete AST (that can be anything from a single line of code, a function, a whole file, or theoretically up to an entire project) and converting it into a bag of paths. A bag, also called multiset, of paths means a list of paths, where the order of those paths does not have any significance and duplicates are allowed. Those paths are the relationship paths covered in subsubsection 4.4.4. Covering too big of a scope may reduce the accuracy as the paths can only cover a limited range in the AST. It is therefore advised to keep the scope as small as possible, which are in this context single methods. The process of converting the paths is visualized in Figure 7. It starts by splitting each of the paths in the bag into three parts: the start token (left violet box), the inner path (orange box), and the end token (right violet box). The two tokens get embedded by the same embedding and are then combined with the separately embedded inner path. This combination then gets fed into a classical neural network (yellow box), with the only special thing being the attention layer (red box) at the end. This attention layer is used for adjusting how much attention a single path in the bag gets. More on the attention mechanism later in subsection 5.3. The output is a vector (green box) that represents all the input paths, and therefore all the code, as a single vector. [AZLY19]

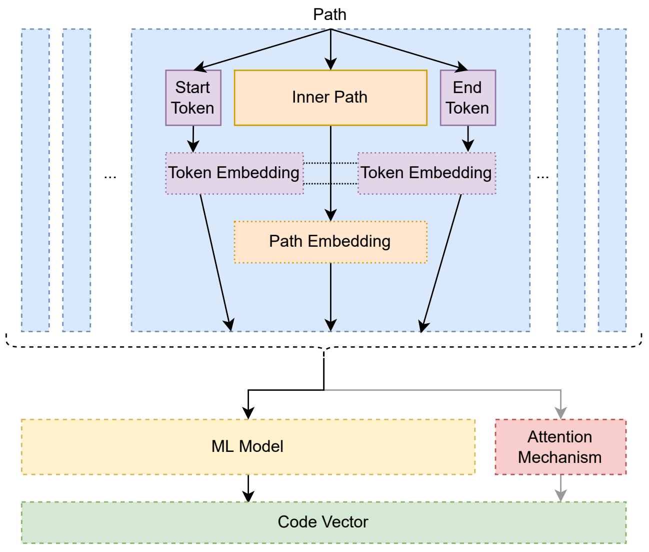


Figure 7. Visualization of Simplified code2vec Model

5.2 Code2seq

Code2seq is the descendant of code2vec. The main difference is that code2seq uses a decoder in the end to output a sequence instead of a single vector.

Internally some improvements are implemented, all of which could also be applied to code2vec. The initial version of code2vec is limited as an encoded inner path is needed instead of a sequence of tokens. In code2seq, the inner path gets encoded by a path encoder using an LSTM[13] layer. Furthermore, all the start and end tokens get split into sub-tokens, as suggested by Allamanis et al. [ABBS15], meaning that words composed of multiple words get decomposed in their individual words. For example, the token "ArrayList" gets split into the sub-tokens "Array" and "List". Those sub-tokens are then encoded and summed to get the final encoded token. This procedure reduces the number of different tokens that need to be embedded, reducing the number of parameters required and making the model, therefore, more efficient. [ABLY19]

5.3 Attention Layer

An attention layer, also called alignment model, is a special layer used to adjust the amount of attention a single path gets. Same as for humans, this is the ability to focus on one thing and ignore others which do not play any important role in achieving the goal we are pursuing. For example, if we want to differentiate a leopard from a tiger, we have to direct our attention towards the dots or the stripes in the fur. The attention layer does that by comparing the parts of the model's input with the model's output and automatically deducing weights according to the importance of a part. The higher the weight, the higher is the importance of that part regarding the output. This weight deduction is done over time during the training of the model. Typically the weights then get normalized to have a total sum of 1, meaning that to each input, a percent value gets assigned according to its importance. The output of the attention layer is a vector that represents the amount of attention that each path gets and the adjusted layer. This can significantly improve the model's quality and has a free extra benefit, namely "explainability". With the information on what input gets how much attention, we can better understand and explain what part of the input had the most impact on the output. For example, when doing sentiment analysis of a text, meaning analyzing whether the text is positive or negative, we can use the attention layer to understand what part of the input had the most impact on the output. More concretely, the sentence "I love you" would have a higher amount of attention on the word "love" than on the word "I" and can therefore easily predict a positive sentiment. In the case of code2vec and code2seq, the attention layer can give us insights about which path in the bag most impacts the output. A visualization of this will be discussed in the next section. [Gér19, BCB, LPM15, Ala, Gra20]

[13]LSTM: Long Short-Term Memory, a particular machine learning type specialized in remembering information of previous items and combining it with the current.

5.4 Revisiting the isPrime-Example

Looking at the initially stated isPrime-example, in Figure 8 we see the AST and its paths together with the attention each path gets. Looking at the picture, we can see that the

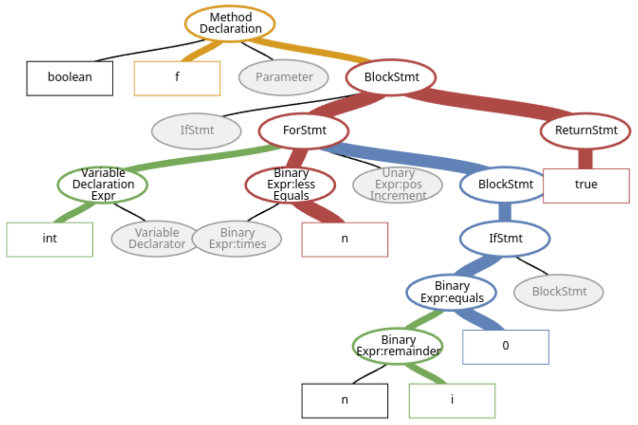
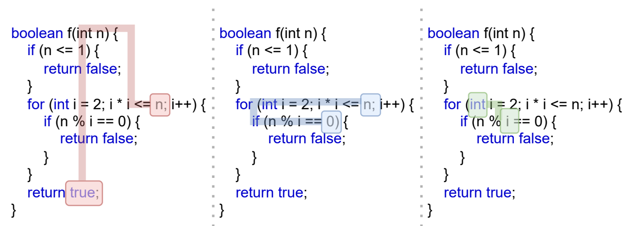


Figure 8. AST representation of the function isPrime (Listing 1). The attention corresponds to the width of the path. Grayed out nodes contain paths that are omitted for readability. [AZLY19]

red path `n`, `<=`, `for statement`, `block statement`, `return statement`, `true` had the most impact on the prediction of the name. If we map this path directly to the code Figure 9, this path mapping would probably not be your first choice as the `n` in



```
boolean f(int n) {
  if (n <= 1) {
    return false;
  }
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      return false;
    }
  }
  return true;
}
```

```
boolean f(int n) {
  if (n <= 1) {
    return false;
  }
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      return false;
    }
  }
  return true;
}
```

```
boolean f(int n) {
  if (n <= 1) {
    return false;
  }
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      return false;
    }
  }
  return true;
}
```

Figure 9. The three most relevant paths of the isPrime example visualized in the code.

the for statement has no big impact on the function returning `true`, other that providing a termination condition. The same goes for the second most important path (marked in blue). For the third most influential path (marked in green), the importance can be seen. The relation between the `int` in the iterator variable declaration of the for loop and the `i` in the remainder calculation is essential. Simply with this information, the possible options for what kind of function this is can be narrowed down to fewer options.

6 Use Cases, Tools, and further Research

To recap, code2vec and code2seq are two slightly different approaches to representing code in a way that can be used for machine learning using AST paths. Code2vec outputs a vector representation of the code, while code2seq outputs a sequence of vectors representing the code. The use case of predicting the method name is the use case used to show off the approaches capabilities in the original paper. But many other use cases can be thought of. Code2seq can be used to create documentation for a given code snippet. Further examples, presented in the paper and on their website[14,15], are:

- predicting similar method names, for example, “size” is similar to “count”.
- combine two methods into a single one. For example, “equals” and “toLower” can be combined into “equalsIgnoreCase”.
- analogies prediction, like the “king-man+woman=queen”-example, for example, “receive” is to “download” as “send” is to “upload“.

6.0.1 astrid. There are already use cases outside the papers. The company JetBrains has created an Intellij IDEA[16] plugin called `astrid`[17] that uses the method name prediction approach to suggest better names directly in your editor. The goal is to make such tools better accessible to everyone and improve the overall code quality. Unfortunately, the project has seen no updates for three years at the moment of writing this.

6.0.2 PSIMiner. `PSIMiner`[18] is another plugin for IntelliJ IDEA crated by the company JetBrains. It is a tool for processing “Program Structure Interface” (*PSI*) trees that are responsible for parsing files and creating syntactic and semantic code models to power many of the platform’s features. [Jet]

6.0.3 astminer. `astminer`[19] is yet another tool by JetBrains. It builds the basis to the two other tools, `PSIMiner` and `astrid`, and also creates the bridge between them and code2vec, or code2seq. It is a library for mining path-based representations of code. Internally it has implementations for many parses like ANTLR, GumTree, or Fuzzy, therefore supporting a wide range of languages. It has configuration options available like storage formats, label extractors, and filters. It makes the process of extracting the initial path representation much more trivial and straightforward. It is either available as a standalone tool or as a library you can access from Kotlin or Java. [KBBB19]

6.0.4 Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding. In “A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding” [PLMH] the authors describe an approach on how to make deep neural networks (*DNNs*) more robust. The approach mutates the input with refactoring tools to create different but semantically equivalent code snippets. Based on their tests, the robustness could be improved by 23% on average. The paper covered code2vec, code2seq, and also CodeBERT.

7 Conclusion

This paper has shown some current state-of-the-art machine learning models and their backgrounds in previous machine learning tasks, especially natural language processing. It also tries to present the differences in an easily digestible way with an incentive of being able to reason about what model should be used for what. Furthermore, it should give valuable insight and understanding of the models to the reader so that it can be used as a starting point for their research.

Even though a lot is already possible, we are still in the early stages of effectively using machine learning for programming languages. There is a lot of research and work going on, with the goal to improve the current solutions, find new solutions for other use cases, and make those machine learning models more accessible to everyone. With the rise of open-source software, a lot of knowledge is available. It just needs to be harvested and used in a helpful way for everyone. In the end, the goal is to make the overall quality of software more stable, more secure, and more consistent.

[14]https://code2vec.org
[15]https://code2seq.org
[16]A popular development environment (https://www.jetbrains.com/idea/)
[17]https://github.com/ml-in-programming/astrid
[18]https://github.com/JetBrains-Research/psiminer
[19]https://github.com/JetBrains-Research/astminer

References

[ABBS15] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting Accurate Method and Class Names. 2015.

[ABK] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to Represent Programs with Graphs.

[ABLY19] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019.

[Ala] Jay Alammar. Visualizing a neural machine translation model (mechanics of seq2seq models with attention). http://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/.

[AZLY] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A General Path-Based Representation for Predicting Program Properties.

[AZLY19] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[BCB] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE.

[DCL+] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova Google, and A I Language. Bert: Pre-training of deep bidirectional transformers for language understanding.

[FGT+] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages.

[Gér19] Aurélien Géron. *Book Review: Hands-on Machine Learning with Scikit-Learn, Keras, and Tensorflow, 2nd edition.* 2019.

[Goo] Google. Overview - seq2seq. https://google.github.io/seq2seq/.

[Gra20] Alex Graves. DeepMind x UCL | Deep Learning Lectures | 8/12 | Attention and Memory in Deep Learning - YouTube. https://www.youtube.com/watch?v=AIiwuClvH6k&ab_channel=DeepMind, 2020.

[HPP+] Marcus Hägglund, Francisco J Peña, Sepideh Pashami, Ahmad Al-Shishtawy, and Amir H Payberah. COCLUBERT: Clustering Machine Learning Source Code.

[Jen21] Raphael Jenni. Machine Learning for Programming Languages - An Overview of Machine Learning for a Software Engineer. 2021.

[Jet] JetBrains. Program structure interface (psi) | intellij platform plugin sdk. https://plugins.jetbrains.com/docs/intellij/psi.html.

[JZL+] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. TreeBERT: A Tree-Based Pre-Trained Model for Programming Language.

[KBBB19] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press, 2019.

[KMBS20] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. 2020.

[LPM15] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective Approaches to Attention-based Neural Machine Translation. pages 17–21, 2015.

[Mig17] Piotr Migda. king - man + woman is queen ; but why ? Intro. https://p.migdal.pl/2017/01/06/king-man-woman-queen-why.html, 2017.

[PDK18] Michael Pradel, Tu Darmstadt, and Germany Koushik Sen. DeepBugs: A Learning Approach to Name-Based Bug Detection. 147:25, 2018.

[PLMH] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. A search-based testing framework for deep neural networks of source code embedding.

[PSD17] Michael Pradel, Koushik Sen, and T U Darmstadt. Deep learning to find bugs. https://github.com/michaelpradel/DeepBugs, 2017.

[Tur37] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[VKM+] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural Program Repair by Jointly Learning to Localize and Repair.