Better Code Representation for Machine Learning

Raphael Jenni
OST Eastern Switzerland University of Applied Sciences
Supervised by Prof. Dr. Luc Bläser
ST 2022

Abstract

Using machine learning for code becomes more and more common. Different approaches based on *paths* or *BERT* are available. This paper focuses on improving parts of the input vector by creating a more compact embedding. Furthermore, it explores and discusses ways to reduce the amount of data inserted into a model when working with code changes. The results presented in this paper show that it is possible to reduce the input data into a latent space, cutting it to half the input data size, representing differences and similarities between code paths in a very compact way while still maintaining an accuracy of 99%. Moreover, it is shown that with proper preprocessing, it is possible to reduce the amount of data inserted into a code changes model by around 84%.

Keywords: Code Representation, Machine Learning, Preprocessing, Change Detection

1 Introduction

Using machine learning for code becomes more and more common. However, it has not yet reached the quality level of other machine learning areas, such as natural language processing or image recognition, mainly because of the limited knowledge of best practices and tools for working with code in a machine learning context. One of those areas where few hard-proven methods are available is the representations of code.

There are multiple studies regarding the representation of source code. Some studies propose handling code as text and with the BERT [DCL+] approach [FGT+, KMBS20, HPP+]. Other studies propose an approach of utilizing the AST in combination with BERT [JZL+] or using paths extracted from the AST [AZLY19, ABLY19, PSD17, PDK18].

This paper focuses on the approach based on Code2Vec [AZLY19] and Code2Seq [ABLY19], and therein mainly on better embedding parts of the input vector in a more compact way and an investigation of the data at hand. Furthermore, ways to reduce the amount of data inserted into a model working with code changes are explored and discussed.

2 Problem Definition

Code2Vec [AZLY19] is a machine learning model that can be used to represent code as vectors. Code is initially represented as a bag of paths, going from one leaf in the AST to another. Such a path is called a code path and is represented in the form of `startTerminal,node|node|...,endTerminal`. The start and end terminals can be identifiers or operators but are never an inner node, such as an expression or statement. The extraction process is visualized in Figure 1. The training

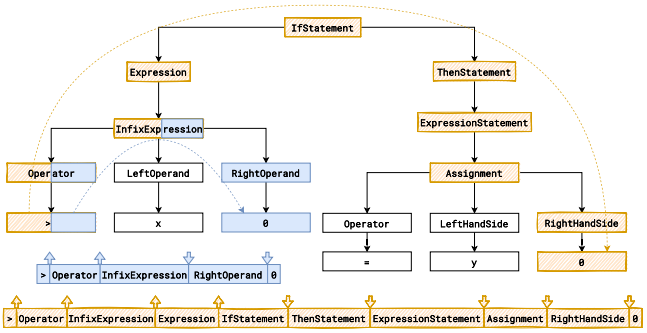

Figure 1. AST to Code Path Example

process (visualized in Figure 2) then follows an embedding step followed by a standard deep learning model with an included attention mechanism. The upper part, colored in blue, represents the embedding of the code paths. This part requires thousands of parameters to be tuned, and involves millions of labeled code samples for it to be learned correctly, even for relatively small models. Doing this leads to prolonged training times and, depending on the goal, to a lot of manual or partly manual labeling of the data. The embedding is the part that should be improved and is the paper's primary focus. The goal is to find an effective embedding

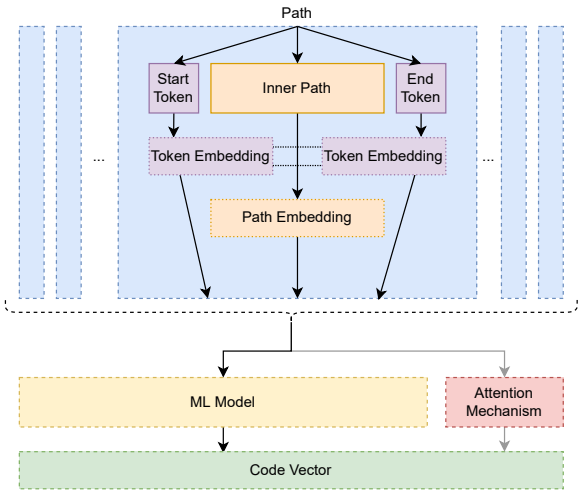

Figure 2. Visualization of Simplified Code2Vec Model

that can represent the whole or parts of such a code path in a small and concise manner. And all this without the need for manual labeling so that it can be applied easily to different programming languages.

Besides finding an effective embedding for representing code in general, another problem is describing code changes. A code change is a change visible in a version control system. However, while code changes contain a lot of helpful information for learning purposes, representing such a change for a machine learning model is not easy. The challenge here lies in finding a way to represent the change, such that the model includes information of what was before the change, what was after the change, and what exactly did change. A task that is sometimes difficult, even for a human. One major factor lies in the amount of data that gets inserted into the model to learn from. Therefore, the goal is to reduce the amount of inserted data in order to focus on the essential things and gain speed for training.

3 Inner Path Embedding

In Code2Vec, the inner path is embedded as part of the training process. This embedding step requires a lot of data for it to be learned. The hypothesis was that pre-training the embedding of the inner paths should yield a much better and more compact representation of the paths itself. With the resulting vector, the model should be able to interpret the inner paths more efficiently and with higher precision.

The executed pre-training process of those inner paths was realized using an autoencoder. The autoencoder's task is to encoded and compacted a path from 9 elements down to 4 and decoded it back to the original version. For controlling outliers, a L1 regularization was added to the bottleneck layer. To use this autoencoder in a model, the decoder part was removed, and the bottleneck layer represented the embedded path.

3.1 Results

The results of the inner path embedding were very promising, with an accuracy of 0.99 or 99% and a loss of only 0.04 on the test data. The embeddings were created based on 2272 Java files from ten major open source projects on GitHub, totaling over 28 million paths. On closer inspection of ten different paths, uniformly selected from different projects, the model was able to embed different paths in the latent space that could be used for similarity comparisons. Ten paths were manually selected that were representative to create a meaningful result, where eight out of the ten paths were genuinely different. Adding more paths, some of them less similar, just confirmed the result and are therefore not further considered in this paper. Figure 4 shows the latent space for the embedding of 8 different paths. The two plots in the first two rows are the same type of paths with a slight variation. The rest of the row pairs are different versions

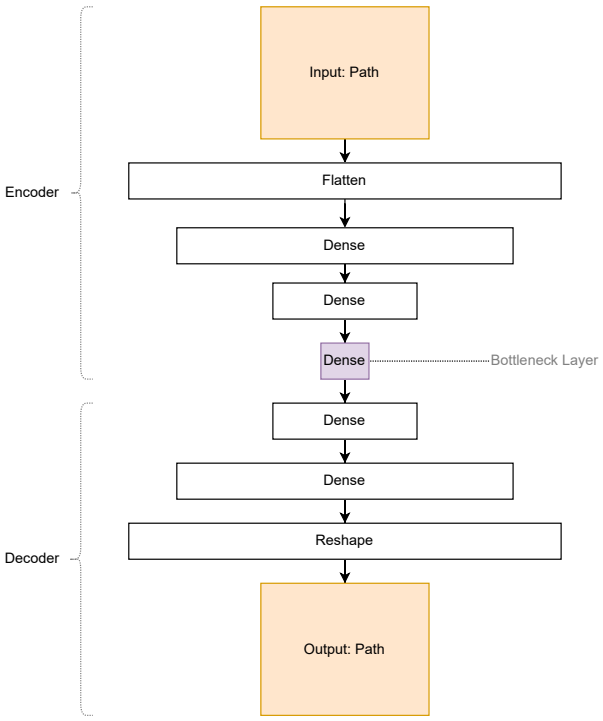

Figure 3. Inner Path Autoencoder Model - Simplified Visualization

of the same types. Each of the paths is listed in Table 1. Interestingly, paths of the same overall type can vary depending on where the path ends. For example, plots 4 and 5 are both for-each loops, meaning loops in Java of the form `for (var item: items) {...}`. However, plot 4 describes the variable declaration in the *for loop* (`var item` in the example), and plot 5 illustrates a method call inside the loop declaration. With regard to the loop structure, plot 5 is more similar to plot 7, as they both contain a method call. This can also be observed when comparing those two plots to each other. Both plots 5 and 7 start and end similarly and therefore are located close to one another in the latent space. Visualizing the latent space in the confusion matrix shown in Figure 5, based on the cosine distance between the latent vectors, shows the similarities very clearly. The closer to 1, visualized with the yellow color, the more similar the two vectors are. Very dissimilar vectors are closer to -1 and are visualized in a darker color, the lowest being -0.4 and visualized in dark purple. This similarity and dissimilarity are an important feature of the inner path embedding. With this, the numbers of the vector get a relevant meaning other than the tokenization id. Furthermore, the embedding has a size that is half the size compared to the original vector and, therefore, also reduces the number of output parameters descending models need to handle. Unfortunately, no other research in this area with available data for comparison could be found.

#	Label	Path
0	If Statement 1	IF|statement|parExpression|expression|primary|IDENTIFIER
1	If Statement 2	IF|statement|parExpression|expression|BANG
2	Variable Assignment 1	LONG|primitiveType|typeType|fieldDeclaration|variableDeclarators|variableDeclarator|ASSIGN
3	Variable Assignment 2	INT|primitiveType|typeType|localVariableDeclaration|variableDeclarators|variableDeclarator|ASSIGN
4	Enhanced For Loop - item declaration	FOR|statement|forControl|enhancedForControl|variableDeclaratorId|IDENTIFIER
5	Enhanced For Loop - expression call	FOR|statement|forControl|enhancedForControl|expression|methodCall|LPAREN
6	For Loop - index declaration	FOR|statement|forControl|forInit|localVariableDeclaration|variableDeclarators|COMMA
7	For Loop - expression call	FOR|statement|forControl|expression|expression|expression|primary|IDENTIFIER
8	Method Declaration	IDENTIFIER|methodDeclaration|methodBody|block|blockStatement|localVariableDeclaration|variableModifier|FINAL
9	Class + Method Declaration	IDENTIFIER|classDeclaration|classBody|classBodyDeclaration|memberDeclaration|methodDeclaration|IDENTIFIER

Table 1. Inner paths example with their corresponding type

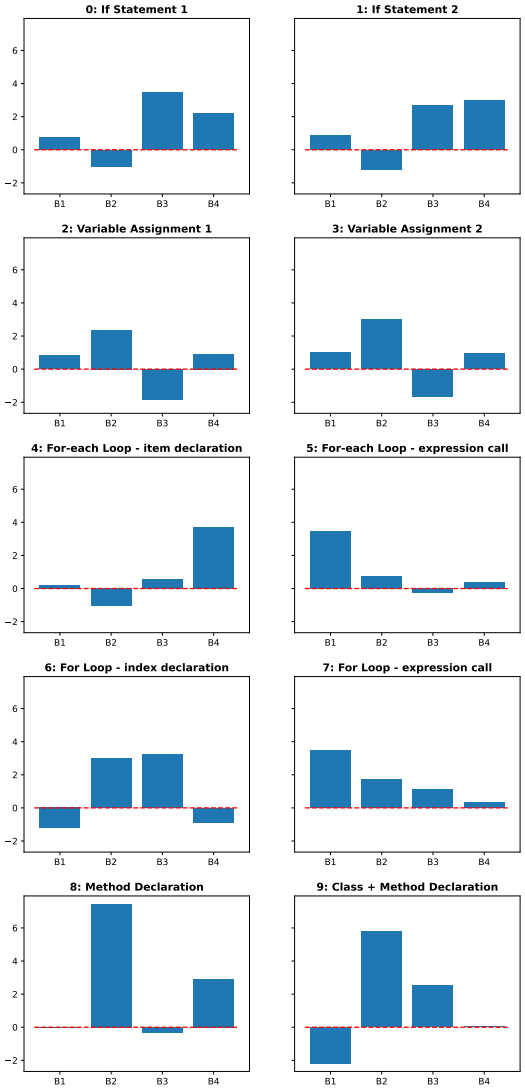

Figure 4. Path embedding of the ten paths of Table 1 plotted according to their embedding vector at each of the four positions.

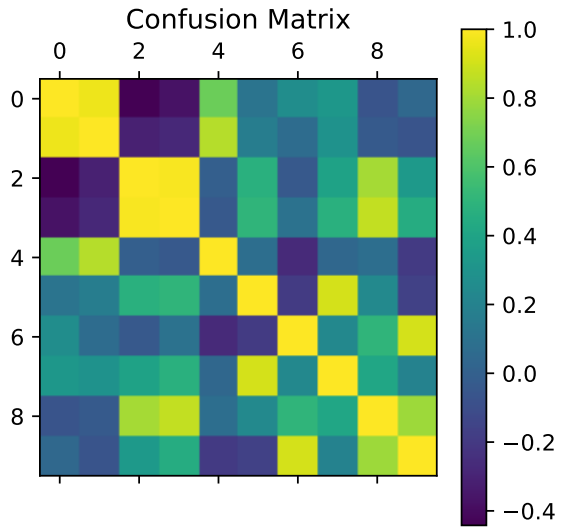

Figure 5. Confusion matrix of the 10 paths of Table 1. 1 means identical, and -1 means dissimilar.

4 Path Data Investigation for Code Change Representations

In addition to the embedding of the paths, an in-depth data investigation was done, trying to reduce the data when mutually comparing code changes. In our case, the data consisted of all the code paths of files affected by the change. The aim was to reduce the number of paths presented to the model in order to reduce unnecessary data and speed up the training process. Moreover, the granularity of the path bags were investigated to analyze the impact of the granularity on the output data. To improve the code representation even further, sub-tokenization for the terminals was analyzed as an additional preprocessing step. This sub-tokenization is inspired by the Code2Seq model and is based on a paper by Allamanis et al. [ABBS15]. Its objective is to separate word combinations into separate tokens and embedding them as

	All Path-Bags	All Path-Bags + Subtokenization	Subtokenization Reduction	Modified Paths-Bags	Modified Paths-Bags + Subtokenization	Subtokenization Reduction	Modification Reduction	Modification + Subtokenization Reduction
# of Paths	39,647,696			28,073,962				
# of Unique Start Terminals	161,332	28,122	83%	48,754	10,971	78%	70%	60%
# of Unique End Terminals	159,987	28,417	82%	48,023	10,965	77%	70%	61%
# of Unique Inner Path Tokens	203			163				
# of Unique Terminals >10 occurences	142'368	23,310	84%	41,712	8,513	80%	84%	64%

Table 2. Comparison of paths bags with and without filtering, and with or without subtokenization.

such. This has the potential to reduce the total number of unique tokens and therefore reduces the required embedding dimension.

When analyzing the data, it came apparent that the granularity of the initially chosen path bags was too high. The bags consisted of all the paths in a single file, but changes mainly affect only parts of a file. By always taking the whole file into account, a lot of duplicate data is present in the before and after version. This data is certainly needed to create an exact understanding of what the code does, but does not play that much of a role when only analysing the changes. Furthermore, we might remove paths that give important context information if we just deduplicate paths. The analysis showed that reducing the granularity from the level of covering the whole version as a single bag of words to the level of method level bags, enabled us to remove a large portion of unchanged paths. Paths of methods that have not seen any modification between the versions before and after the code change have been removed, reducing the size of the bag by 30%. Applying this filtering process, the number of unique terminals in the bag even decreased by 83%.

The sub-tokenization of the terminals also yielded some significant size reductions. Analyzing the number of different terminals in the path bags with over ten occurrences showed that the sub-tokenization resulted in a reduction of 84%. All the numbers and comparisons are shown in Table 2. It is worth noting that due to the already high reduction of only taking the modifications, the further reduction by subtokenizing, shown in the last two columns, is minor. The total reduction from no filtering to modification and subtokenization filtering falls between 93% and 94%.

5 Conclusion

Applying deep learning to code changes could have high potential for various practical use cases in software engineering, such as inferring bugs, clustering semantic changes, or trend recognition. For this purpose, using paths for learning embeddings seems to a very promising approach, especially if pre-processing them and removing duplications, although other AST based approaches are equally possible. Using pre-trained embeddings can reduce the data required for further training and speed up the training process due to the fewer training parameters present. Pre-training a model with an autoencoder model eliminates the need for labeled data and can therefore be done on a very large scale. The inner path embeddings showed that it is possible to reduce the size by 50% of the input data while maintaining an accuracy of 99% and a loss of only 4%. Although the inner path embeddings reduce the path length by half, embedding the whole code path is still required, as the start- and end-terminal need to be included. Further research must be done to find a good embedding that can represent the entire code path and the whole bag of paths.

Working with code changes is very difficult because of the not yet existent solid code representation. Applying pre-processing techniques can reduce the noise around data and push the process in the right direction. Filtering duplicate paths between the two versions of a commit can already reduce the amount of data by 30%. Sub-tokenization reduces the number of terminals further by around 75%.

More research is required to realize meaningful and practical code analysis use cases with deep learning. There the practical use of the here presented compacter embeddings can be confirmed. Such a use case could be adjusting existing models, like the Code2Vec and Code2Seq models, to use this first embedding step and train the model with fewer data. Using more advanced code understanding techniques, which will undoubtedly improve in the near future, could lead to new and different insights.

6 Related Work

As already mentioned in section 1 multiple studies regarding the representation of source code based on BERT [DCL+] or paths are available.

A study covers detecting one of three commit types with deep learning [LY]. The approach uses a combination of static AST analysis, keyword analysis and deep learning to detect the type of commit. The authors also open sourced the dataset used for training[1].

The paper "CC2Vec: Distributed Representations of Code Changes" by Hoang et al. [HKLL20] created a neural network model that learns code representations of code changes with

[1] https://zenodo.org/record/835534

the help of the corresponding log messages. This representation is aimed to represent the semantic intent of a code change. They tested the model on three tasks: *Automatic Log Message Generation*, *Bug Fixing Patch Identification*, and *Just-in-Time Defect Prediction*. *Bug Fixing Patch Identification* is based on PathNet [HLO+] and should help backport bugfix patches to older versions. *Just-in-Time Defect Prediction* should provide early feedback if a path has a defect.

Using more compact embeddings is a research task in many areas of machine learning. Autoencoders are one possibility, where besides the advantage of being an unsupervised process, the resulting bottleneck layer is a compact representation of the input data. Autoencoders have been first introduced in the 1980's [RHW86]. Nowadays, they are used in countless different applications and in many forms. The paper "Autoencoders" [BKG] covers and explains different types of autoencoders.

References

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

[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.

[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.

[BKG] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders.

[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.

[HKLL20] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. CC2Vec: Distributed Representations of Code Changes. 2020.

[HLO+] Thong Hoang, Julia Lawall, Richard J Oentaryo, Yuan Tian, and David Lo. PatchNet: A Tool for Deep Patch Classification.

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

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

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

[LY] Stanislav Levin and Amiram Yehudai. Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing Source Code Changes.

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

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

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. 1986.