Who wrote that?

Richard Čerňanský
Ing. Juraj Perík
Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
`xcernansky@stuba.sk`

November 23, 2024

Abstract

abstract abstract abstract abstract

Chapter 1

Introduction

1.1 Motivation

In this thesis, we aim to analyze the performance of our re-implemented code2vec CNN model, which utilizes an attention mechanism to process concatenated inputs to classify function under a certain name. The input to this model is derived from node-to-node features of functions' Abstract Syntax Trees (ASTs). The evaluation focuses on source code snippets written in the C programming language, sourced from datasets of competitive programming events.

1.2 Domain Analysis

1.3 Source code feature extraction

Source code is essentially just a text representation of the algorithm it represents for the machine. In source code analysis, what we need is concise method on how to extract syntactic, semantic and contextual information that defines such code snippet. There are many types of representation methods like token-based, tree-based and graph based. [5]. Watson in his book [7] proposes similar division, he adds character level features and provides us with exact feature definitions for each category. What we are interested in is the tree-based representation.

This section answers the following questions:

1. What techniques are utilized to process source code for extracting its unique information?
2. What is an Abstract syntax tree?
3. What are the possibilities for extracting features from an AST?
4. How do we utilize the information stored in a function's AST?
5. Why is this approach used?

1.3.1 What techniques are utilized to process source code for extracting its unique information?

1.3.2 What is an Abstract syntax tree?

Abstract syntax tree (AST) is a tree-based representation of source code, where nodes represent various elements of the source code and paths represent relationships of the structure. AST is an inseparable part of compilation process for many reasons. It is used as an intermediate representation of program utilized for optimization and generation of machine code. Nodes of the tree can be either internal (non-leaf) or leafs. Internal nodes define the program's constructs or operations for their children. Leaf nodes store the actual textual value. Let's see the following definition.

Definition 1 *[6]* ***Abstract Syntax Tree (AST).*** *An Abstract Syntax Tree (AST) for a method is a tuple $\langle N, T, X, s, \delta, \phi \rangle$ where N is a set of non-terminal nodes, T is a set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \to (N \cup T)^*$ is a function that maps a non-terminal node to a list of its children, * represents closure* *operation, and $\phi : T \to X$ is a function that maps a terminal node to an associated value. Every node except the root appears exactly once in all the lists of children.*

Example of AST
Below is an example of a simple C function and its corresponding Abstract Syntax Tree (AST):

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

The corresponding AST is structured as follows:

```
TranslationUnit
|--FunctionDefinition 'int add(int a, int b) {'
   |--BasicTypeSpecifier 'int'
   |--FunctionDeclarator 'add('
   |  |--IdentifierDeclarator 'add'
   |  |--ParameterSuffix '(int a, int b)'
   |     |--ParameterDeclaration 'int a'
   |     |  |--BasicTypeSpecifier 'int'
   |     |  |--IdentifierDeclarator 'a'
   |     |--ParameterDeclaration 'int b'
   |        |--BasicTypeSpecifier 'int'
   |        |--IdentifierDeclarator 'b'
   |--CompoundStatement '{ return a + b; }'
      |--ReturnStatement 'return a + b;'
         |--AddExpression 'a + b'
            |--IdentifierName 'a'
            |--IdentifierName 'b'
```

1.3.3 What are the possibilities for extracting information from an AST?

Features divide into 4 main kinds according to the nature of the information extracted from the Abstract Syntax Tree (AST):

- **Structural**
  Structural features capture the structural complexity of AST. These are often some numerical quantitative values like depth of the graph, number of nodes, or average branching factor of internal nodes.
- **Semantic**
  Semantic features reflect the semantic information encoded in AST. It could be as simple as the distribution of node kinds (that are defined by the compiler that creates the AST) or the distribution of distinct root-to-leaf paths.
- **Syntactic**
  Syntactic features describe the paradigm in which the source code is written and the logical complexity of the program. These might include the number of functions or functor structures, or the number of control flow units.
- **Combined**
  Combined features are use-case specific, and it is up to the data analyst to design the best fit solution for information extraction using the combinations and alternations of the aforementioned strategies. Combined features are also the ones that are usually used when tackling real-world problems like source code authorship attribution or function name classification.

1.3.4 How do We utilize the information stored in a function's AST?

The problem of function name classification requires such information extraction design to capture all three of the structural, semantic and syntactic features in a number vector that could be somehow fed into a neural network classifier. Code2vec [2] proposes solution design that extracts so called 'path-contexts' which are encoded into vector space using the embedding values of its components. Let's see the following definitions.

Definition 2 ***Path context in AST*** *[3] Path context is a triplet consisting of path between two leaves in a tree and the starting node's data (token value) and the ending node's data structured like this: $\langle start\_node.data, path\_arr, end\_node.data \rangle$. So essentially what path context represents is the connected leaf nodes and the path that connects them.*

Definition 3 ***Path in AST*** *Path is a sequence of connected nodes, specifically, what interests us is the sequence of types (kinds) of the AST internal nodes between two distinct (for our purposes leaf but can be any two nodes) nodes.*

Example of path context
Below is an example of a path context between a node with data 'a' and 'BasicTypeSpecifier' node 'int' (of the branch with data 'int b') derived from the Abstract Syntax Tree (AST) that we constructed earlier in the example:

```
⟨ 'a', (IdentifierName (up), AddExpression (up), ReturnStatement (up),
       CompoundStatement (up),
       FunctionDefinition (down), FunctionDeclarator (down),
       ParameterSuffix (down), ParameterDeclaration (down),
       BasicTypeSpecifier (down), 'int' ⟩
```

Source code represented as a bag of path contexts

Finally, after extracting the path contexts between all the distinct nodes, we have a representation almost suitable as input for the neural network classifier. The only step that is left is to encode (convert) the context into numerical vectors. For this, we will use mapping function i.e. vocabularies for each of unique leaf_node.data, tuple(path_from_node_to_node), and also for the output - all the names of our training dataset functions to their unique indices (if the name of function in test dataset was not seen in training dataset we ignore such prediction of classifier). This encoding allows us to map the indices to the embedding space. The embeddings of the components of the path context are then concatenated and used as raw numerical value input for the neural network.

1.3.5 Why is this approach used?

It has shown that the tree traversal that is incorporated in the path-context successfully captures the structural, syntactic and semantic value of the source code [1]. The problem of assigning function names using code2vec model can be stated as learning the information connection between the path contexts and the function names. It is important to note that **not all path context contribute with the same informative value** to the resulting prediction of the classifier. That is why **Attention mechanism** [4] is added to the network to give the path contexts in the function's bag their corresponding weight when predicting the name. This way we can focus on the important path contexts that differentiate syntactically closely related functions from one another.

Chapter 2

Re-implementation of code2vec CNN classifier

2.1 Data Exploratory analysis

The dataset of functions used to train the model consists of extracted C language solutions of the Google Code Jam and Codeforces programming contests. The datasets of Google Code Jam is available on this link and Codeforces on this link.

2.1.1 The Nature of the Data

As the dataset originates from competitive programming contests, its characteristics align with the expected patterns of this domain. However, there are certain limitations and challenges that must be considered when working with such data. Based on our observations, the following points should be carefully evaluated:

1. **Duplicate Entries:** Checking code snippet strings for duplicates to ensure a single submission is not included in the dataset multiple times.
2. **Outliers:** The presence of abnormal functions from source code submissions may impact the analysis and model performance. These include:
   (a) **Extremely Short or Long Functions:** Functions that are either unusually short or excessively long may introduce noise into the dataset and require special handling. For model predicting function names, it would be the best if function follows single-responsibility principle. (some ref would be appropriate i guess)
   (b) **Poorly Named Functions:** Inconsistent or non-descriptive function names can reduce the information gain for the model and negatively affect results. This task is, however, very difficult to address, as it is challenging for a machine to evaluate the descriptive meaning of a function name.
   (c) **Highly Complex Functions:** Functions with excessive complexity can be challenging to analyze and may contain logical circles or recursive patterns in extended representations like call graphs or control flow graphs. While an AST is acyclic by definition, complex functions with recursion, indirect function calls, or unconventional control flows may create logical loops that complicate the analysis and interpretation.
   (d) **Functions Violating the Single Responsibility Principle:** Functions that perform multiple unrelated tasks with name that does not capture all of them can be problematic for the model to evaluate the function correctly which may lead to decreased accuracy.
   (e) **Functions with Rarely Occurring Names:** Rare function names present a challenge for the model, as the limited number of occurrences provides insufficient examples for the model to effectively distinguish them from similar functions with different names.

An initial analysis of the function data can involve visualizing quantitative features, such as the distribution of function names.

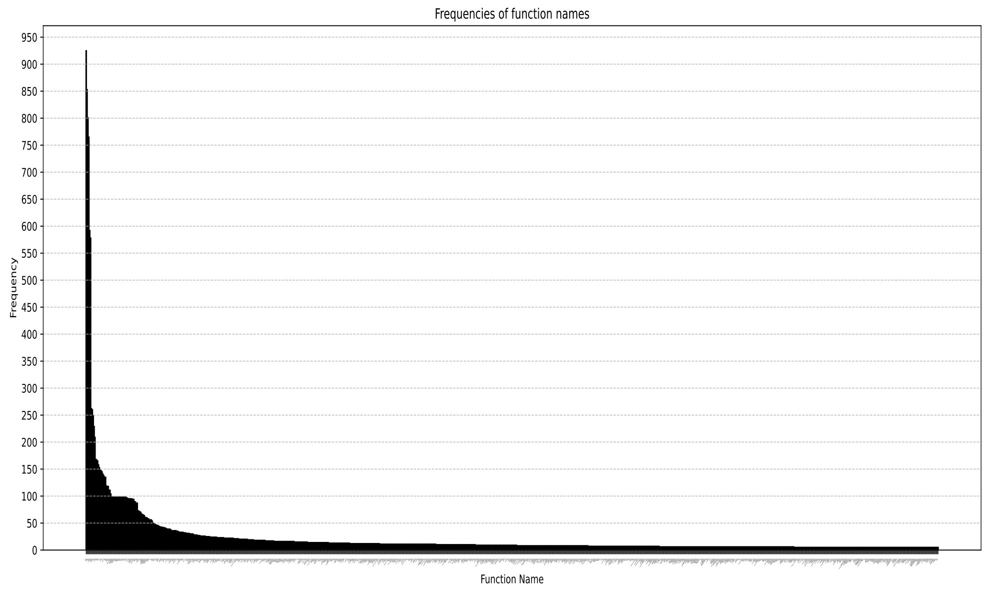

Figure 2.1: Distribution of function name frequencies.

Function name frequency

The plot reveals a long-tailed distribution of function name frequencies, with a few names occurring very frequently and many appearing rarely, but this view is inherently influenced by sorting the data by frequency. This skewed distribution highlights the dominance of common names and the sparsity of unique ones, which could introduce bias and pose challenges for machine learning models. Consequently, it is not possible to see the true underlying distribution of the data. To evaluate whether the distribution is normal (or follows another pattern), we would need to analyze the raw, unsorted frequency data through statistical normality tests.

Shapiro-Wilk Normality Test Results The Shapiro-Wilk test was applied to detect whether the data follows a normal distribution. The results are summarized below:

Table 2.1: Shapiro-Wilk Test Results for Function Name Frequencies

Statistic	**P-value**	**Conclusion**
W = 0.2354	1.5252×10^{-52}	Data is not normally distributed ($p < 0.05$)

Function length measured in tokens

Another basic feature that describes characteristic of data is the length of a function. There are multiple ways to represent such length when observing source code features, for now, we chose to just split by the whitespace characters and count the number of words.

Table 2.2: Kolmogorov-Smirnov Test (for normality) Results for Function Name Frequencies

Statistic	**P-value**	**Conclusion**
KS = 0.2492	0.0000×10^{0}	Data is not normally distributed ($p < 0.05$)

2.1.2 Data preprocessing

Having the data available in .csv tables format, we must have first deal with the filtering the necessary rows (solutions written in C language), transforming this code to ASCII AST using psycheC compiler. From the ASCII

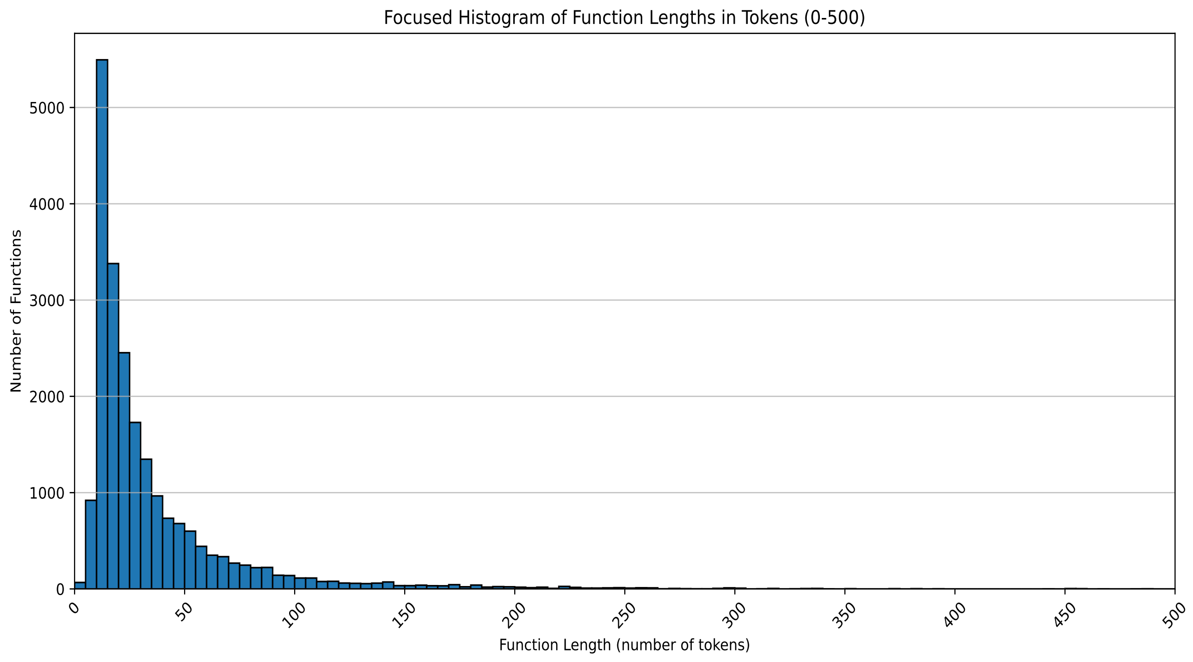

Figure 2.2: Distribution of function lengths measured in tokens.

output of the command *cnip -l -C -d file_path* is actual AST constructed and each function retrieved one at a time and saved (with corresponding metadata) in .ndjson as a single json line.

In the entire training pipeline, there are multiple stages where some of the above-mentioned challenges are resolved.

1. **Duplicate Entries:** These are checked in the initial stage of loading the data from task submissions in .csv into .ndjson files of function ASTs (one per line)
2. **Poorly Named Functions:** In the script *home/training_pipeline/analysis/data_drops_and_analysis.py* we perform dropping of functions that we considered poorly named after seeing the frequency distribution of the names.
3. **Functions with Rarely Occurring Names:** In the script *home/training pipeline/analysis/data drops and analysis.py* we also perform dropping of functions whose names have lower frequency than 5 for stratification purposes (and also model performance).
4. **Extremely long or short functions:** We are interested on how the model will perform when the data is cleaned from outliers in the context of function length in tokens.

2.2 Model Architecture

In this section, we would like to provide a clear explanation of the model that was used to process the data and perform the predictions that are observed. This includes a detailed description of the architecture, the reasoning behind the choice of components, and the methodology employed for training and evaluation.

Model: ”functional”

Layer (type)	Output Shape	Param #	Connected to
value1_input (InputLayer)	(None, 723003)	0	-
path_input (InputLayer)	(None, 723003)	0	-
value2_input (InputLayer)	(None, 723003)	0	-

Layer (type)	**Output Shape**	**Param #**	**Connected to**
value_embedding (Embedding)	(None, 723003, 128)	1,081,216	value1_input[0][0], value2_input[0][0]
path_embedding (Embedding)	(None, 723003, 128)	3,828,992	path_input[0][0]
concatenate (Concatenate)	(None, 723003, 384)	0	value_embedding[0][0], path_embedding[0][0], value_embedding[1][0]
dense (Dense)	(None, 723003, 128)	49,280	concatenate[0][0]
dense_1 (Dense)	(None, 723003, 1)	129	dense[0][0]
weighted_context_layer (WeightedContextLayer)	(None, 128)	0	dense_1[0][0], dense[0][0]
tag_embedding_matrix_layer (TagEmbeddingMatrixLayer)	(None, 740)	94,720	weighted_context_layer[0]
softmax (Softmax)	(None, 740)	0	tag_embedding_matrix_layer

Total params: 5,054,337 (19.28 MB)
Trainable params: 5,054,337 (19.28 MB)
Non-trainable params: 0 (0.00 B)

2.2.1 Input layer

2.3 Training pipeline

Chapter 3

Testing

Heatmap Analysis:

The heatmap visually represents the average performance metrics (precision, recall, F1-score). Look for patterns such as classes with consistently low or high scores to identify well-performing or problematic categories.

Key Metrics:

Precision: High values indicate fewer false positives. Classes with low precision might be confused with other classes. Recall: High values mean fewer false negatives. Low recall indicates the model struggles to detect certain classes. F1-Score: Balances precision and recall. Low scores suggest overall poor performance for those classes.

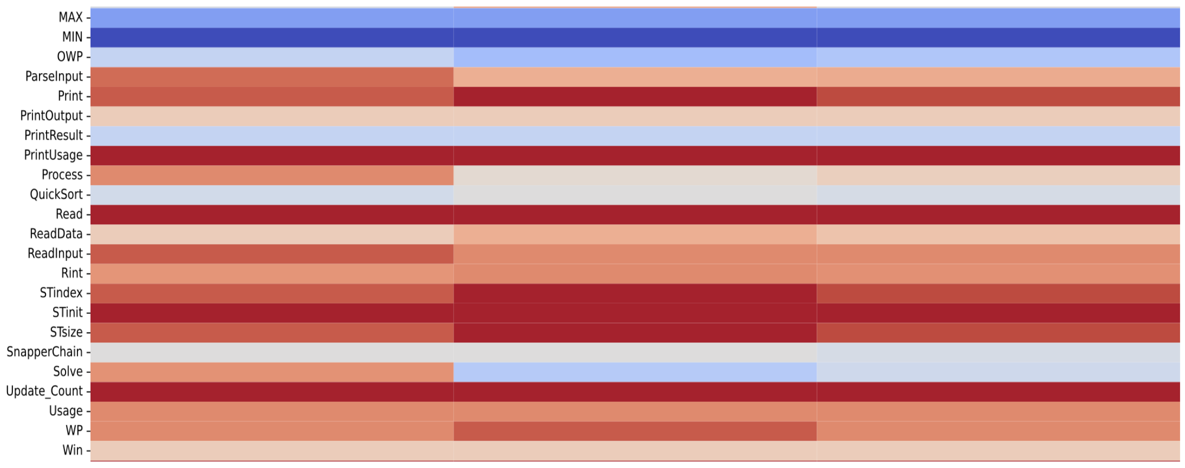

Figure 3.1: Snippet from heatmap of classes of names.

Bibliography

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 404–419, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[3] Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. Authorship attribution of source code: A language-agnostic approach and applicability in software engineering, 2021.

[4] Andrea Galassi, Marco Lippi, and Paolo Torroni. Attention in natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(10):4291–4308, 2021.

[5] H. P. Samoaa, F. Bayram, P. Salza, and P. Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *Software: Practice and Experience*, June 2022.

[6] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and Zhenyu Chen. Abstract syntax tree for programming language understanding and representation: How far are we?, 2023.

[7] Daniel Watson. Source code stylometry and authorship attribution for open source. 2019.