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

Chapter 1

Introduction

1.1 Motivation

In this thesis, we aim to analyze the performance of our re-implemented code2vec NN 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. [6]. Watson in his book [8] 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 *[7] **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, $\delta : N \rightarrow (N \cup T)^*$ 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 NN classifier

2.1 Exploratory data 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:** In programming competitions, where competitors often use template helper functions, it is important to check function code strings for duplicates to ensure that identical functions are not included multiple times in the dataset.

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. When fetching the code snippets from the datasets, we applied a filter to filter out functions with names like 'main' and 'solve', because these names unquestionably do not provide any insight into the functionality.

   (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 names that do 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 distributions of quantitative features (f.e. function names).

Function name frequency

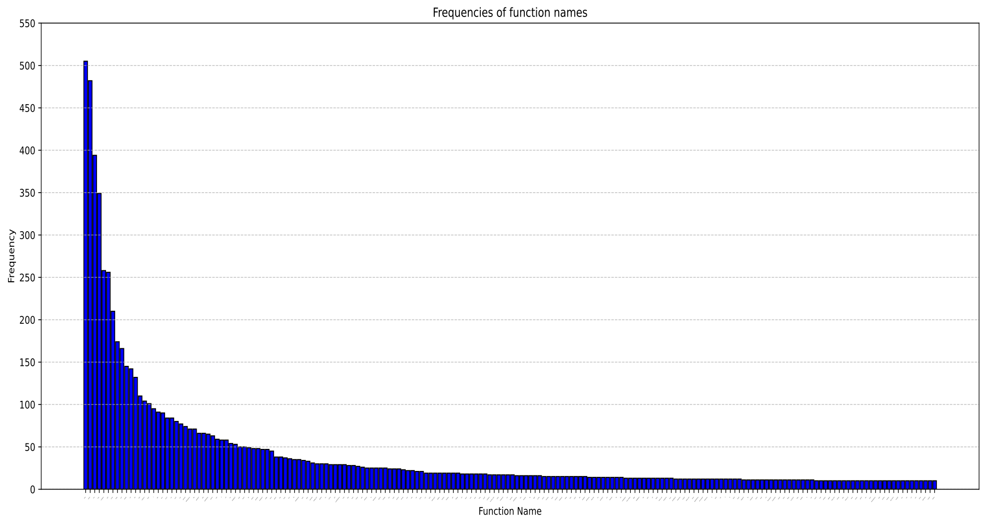


Figure 2.1: Distribution of function name frequencies.

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 ($p < 0.05$)
Mean	**Standard Deviation**	**—**
22.7430	65.4305	—

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.

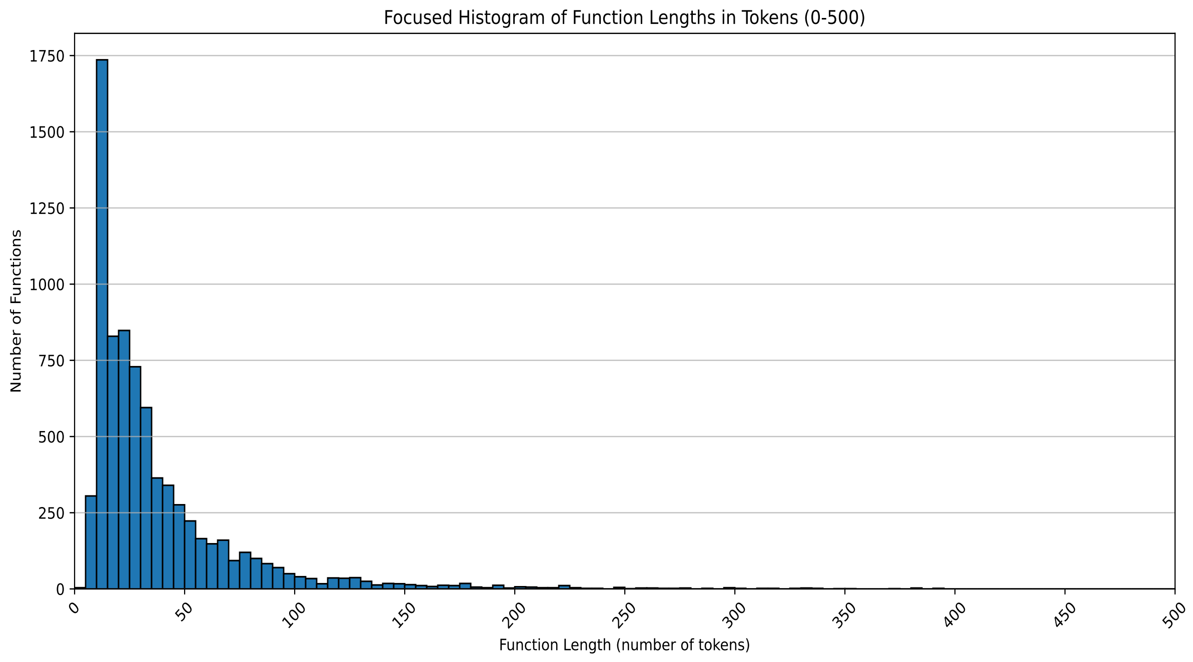


Figure 2.2: Distribution of function lengths measured in tokens.

Table 2.2: Kolmogorov-Smirnov Test Results for Number of Tokens

Statistic	**P-value**	**Conclusion**
KS = 0.2482	0.0000×10^0	Data is not normally distributed ($p < 0.05$)
Mean	**Standard Deviation**	**—**
36.6910	45.8380	—

Function's AST Depth

The distribution of AST depth is a useful metric for describing the data to understand the structural complexity of the dataset 1.3.3.

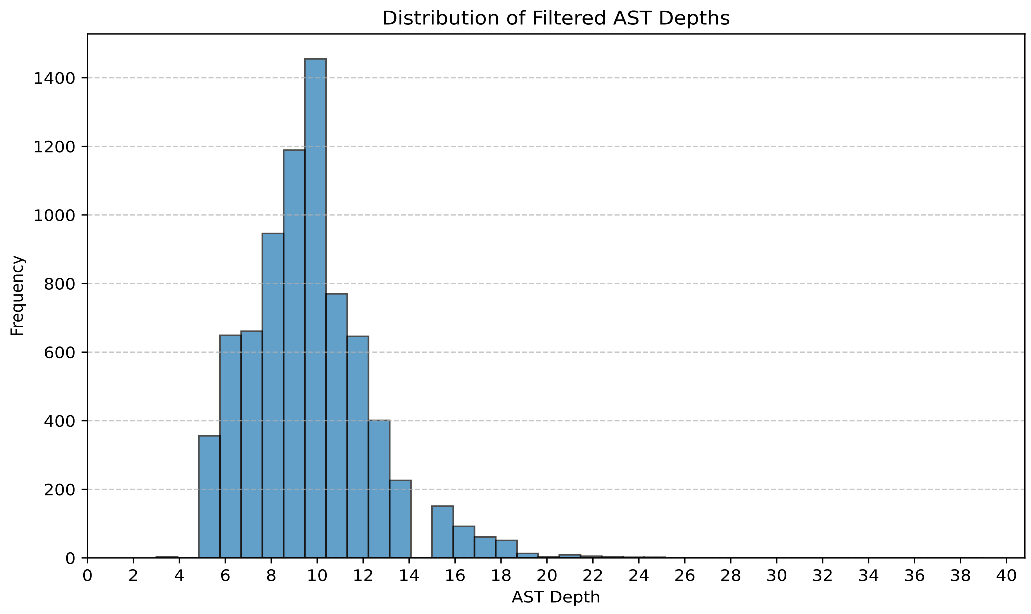


Figure 2.3: Distribution of function's AST depths.

Table 2.3: Kolmogorov-Smirnov Test Results for AST Depths

Statistic	**P-value**	**Conclusion**
KS = 0.1371	0.0000×10^0	Data is not normally distributed ($p < 0.05$)
Mean	**Standard Deviation**	**—**
9.3679	2.8933	—

Function's AST Number of nodes

Similar to the AST Depth, AST's Number of nodes also retains information about structural complexity.

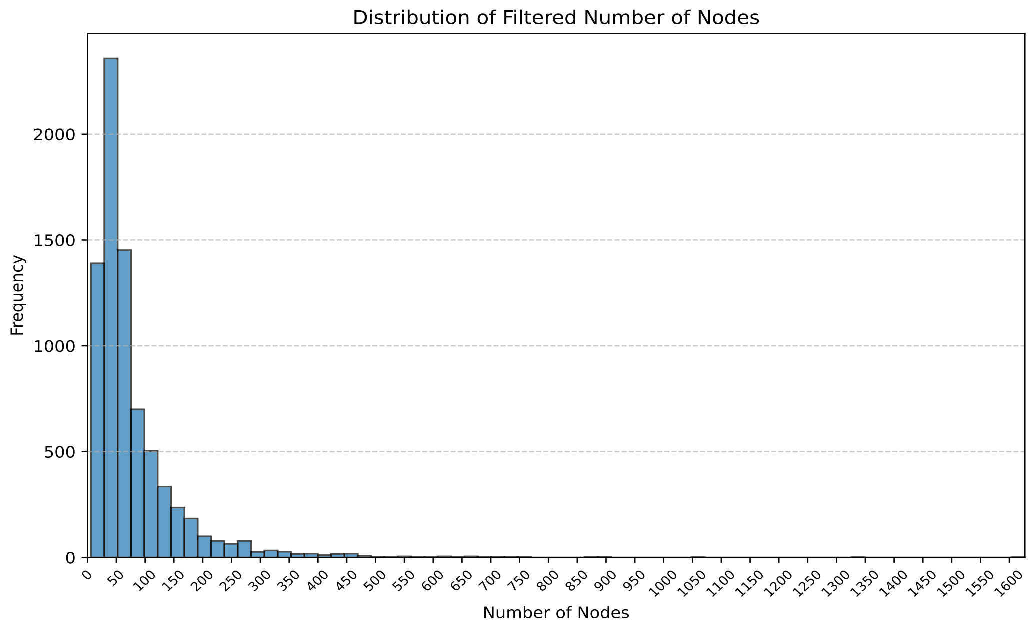


Figure 2.4: Distribution of function's AST Number of nodes.

Table 2.4: Kolmogorov-Smirnov Test Results for Number of Nodes

Statistic	**P-value**	**Conclusion**
KS = 0.2484	0.0000×10^0	Data is not normally distributed ($p < 0.05$)
Mean	**Standard Deviation**	**—**
76.7331	96.9949	—

Key statistics for path-context based approach

When working with NN classifier, we need to know the exact dimensions for our inputs' embedding matrices. Following metrics describe the sizes of vocabularies that encode the natural-language based features to the numerical ones.

Table 2.5: Key statistics for path-context based approach

Metric	**Count**
Number of tags (function names) in total	970
Number of unique terminal node Data	10055
Number of unique node-to-node paths	40520
Number of unique path-contexts	723003

EDA conclusion

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 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 an 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”

As stated before, we decided to use the Attention based neural network classifier to predict C function name from its source code. The model is constructed using tensorflow.keras framework.

2.2.1 Inputs

The model has 3 input layers. One for each part of the path context 1.3.4 and they are represented as vectors of indices to their respective vocabularies. The shape is decided by the maximum number of distinct path contexts from the set of all functions in our dataset (the missing values are padded with zeros).

2.2.2 Input embeddings

For each input layer there is also corresponding embedding layer and the embeddings are obtained using the numerical index representations. Then they are concatenated into one single vector of dimensions (1, 3d) where d is the dimension of single embedding row.

2.2.3 Dense layer

The concatenated vector is then multiplied with weight matrix of dimensions $(y, 3d)$ to obtain vector of dimension y, where y is the size of tags set. Tanh function is applied to this transformed vector to introduce non-linearity.

2.2.4 Attention-weight mechanism

Attention weight vector is then applied to get the portion (importance) of each path context by multiplying the attention weight a_i with path context c_i to get the vector v_a.

2.2.5 Softmax

The final vector of probabilities is then obtained by multiplying each tag's entry in embedding layer of tags $(y, 1)$ with the vector v^T $(1, y)$ and applying the softmax over all the tags in tag set.

This foward process is depicted in the following flowchart:

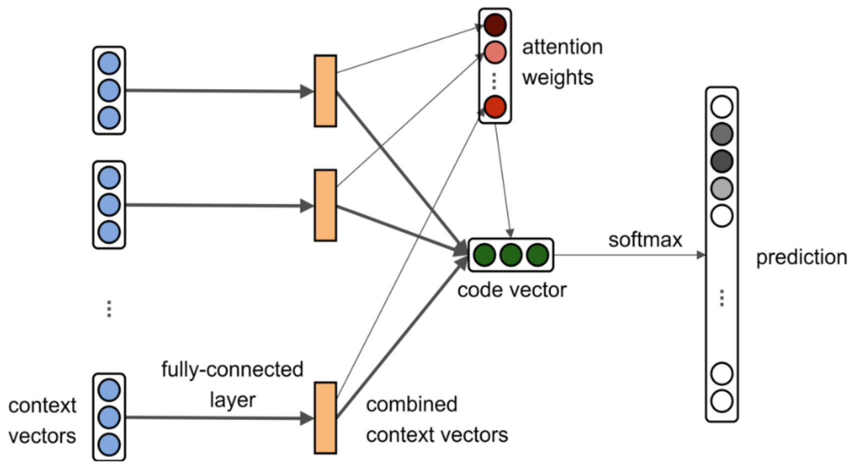


Figure 2.5: Code2vec [2] neural network model flowchart.

The model architecture is displayed in detail below in the table.

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	-
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)

2.3 Training pipeline

2.3.1 Directory structure

The training pipeline of this project is defined in the directory `./training_pipeline`:

```
training_pipeline
|-- analysis
|-- data_ndjson
|-- extract_functions
|  |-- psychec
|  |-- tmp
|  |-- AsciiTreeProcessor.py
|  |-- main.py
|  |-- Node.py
|  |-- NodeTree.py
|-- trained_models
|  |-- vocabs_*.pkl
|  |-- trained_model_*.h5
|-- aggregated_avg.py
|-- AttentionNNClassifier.py
|-- consts.py
|-- generate_vocabs.py
|-- NodeToNodePaths.py
|-- stratifiedKFold.py
|-- test_one.py
|-- testing_model.py
|-- train_valid_strat.py
```

Figure 2.6: Directory structure of the training pipeline (folders in bold).

- **analysis :** This folder contains folders for EDA, training, validation and testing metrics plots. There is also a script for data preprocessing.
- **data_ndjson :** This folder serves as the storage for all the .ndjson files containing AST data, whether it is the initial datase, preprocessed dataset or temporary stratificated datasets that are created during the pipeline.
- **extract_functions :** Here the initial datasets of ASTs (and their metadata) constructed from C source code snippets from GCJ and Codeforces are extracted using the script *main.py*. The other files are definitions of classes and helper functions for this extraction
- **extract_functions/psychec :** This folder is cloned Psyche-C project [5]. Psyche-C is a compiler frontend for the C language that provided us with AST construction.
- **trained_models :** As you can see in the figure 2.6 the files in this folder with asterisk in their name a represent all the vocabs and trained models.

2.3.2 Pipeline Execution: Workflow and Training Process

This subsection explains the data pipeline through numbered steps to illustrate the complete function name classification process, from data extraction to model evaluation. These are the steps in the workflow:

1. Build and compile `psychec` for C code snippet AST generation. It is able to provide structural inference for incomplete code.
2. Run `extract_functions/main.py` to generate `.c` functions ASTs in `.ndjson` format (AST features as metadata included).
3. Execute `data_drops_analysis.py` to drop functions with names occurring ≤ 10 times and filter out poorly named functions (ref. 2b). This script also generates the EDA for the selected features 2.1.
4. Execute `stratifiedKfold.py` to perform stratified splitting of the dataset into training and validation sets for NUM_FOLDS=5. This is the main execution script as it calls the following scripts for each fold to train and test the model.
5. Use `generate_vocabs.py` to generate vocabularies from the stratified train and validation sets and set them up for the current fold model.

6. Execute `test_valid_strat.py` for another stratification for test and validation sets.
7. Train the fold model using `AttentionNNClassifier.py`.
8. Run `testing_model.py` to evaluate the trained model. The evaluation consists of classical metrics such as accuracy, precision, and recall. Additionally, we computed accuracy across different bins based on the following keys:
    - `num_tokens_50_bin_accuracies`
    - `num_tokens_20_bin_accuracies`
    - `ast_depth_5_bin_accuracies`
    - `ast_depth_2_bin_accuracies`
    - `num_nodes_50_bin_accuracies`
    - `num_nodes_20_bin_accuracies`

    We also generate a heatmap for examining average metrics per class (precision, recall, F1 score). The snippet from the heatmap will be displayed later in the chapter Testing 3.
9. After all fold processed, run `aggregated_avg.py` to aggregate the results from all folds.

Chapter 3

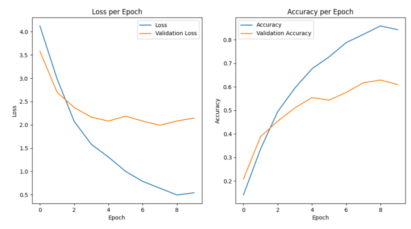
Training and testing evaluation

3.1 Fundamental Data Preprocessing

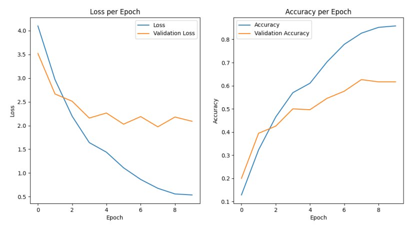
Initally, we droppe functions with names occurring ≤ 10 times and filter out poorly named functions (ref. 2b). Duplicate function strings were also removed to account for redundancies commonly occuring in competition-style programming, where template-based helper functions are frequently reused. The dataset is not yet preprocessed in terms of outliers (length, or tag semantic value).

3.1.1 Training

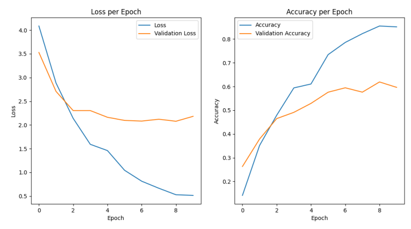
To evaluate the model, we used 5-fold cross-validation on our dataset of C functions from the GCJ and Codeforces competitions. The dataset is not yet preprocessed in terms of outliers (length, or tag semantic value). Batch size was set to 4. The following charts depict learning curves for train and validation accuracy for each fold:



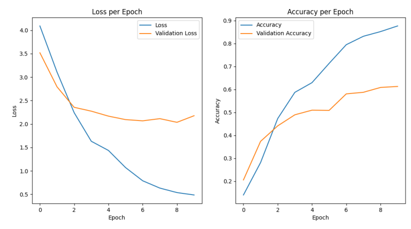
(a) Fold 1



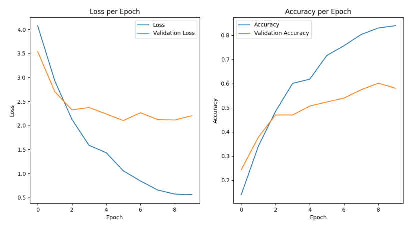
(b) Fold 2



(c) Fold 3



(d) Fold 4



(e) Fold 5

Figure 3.1: Learning curves for all 5 folds.

From the plots, we can say that the model started to deviate training and validation accuracy after 3 epochs achieving overfit and completely stops improving after 8 epochs. (the model memorizes patterns and noise from the training data and fails to generalize on unseen data). Learning patterns in all 5 folds are similar and metrics in fig. 3.2 showed low standard deviation (around 1-2%), this indicates consistent performance across different data splits, suggesting the model is robust. However, validation loss and validation accuracy curves significantly diverge from training curves. We suppose, this is due to following:

- Our insufficient training data which is limited as we are limited by our datasets and only C programming language functions.
- Non-uniformly distributed data mainly in terms of the names of functions (but also due to other features see 2.1.1)
- Noise in the data - f. e. two almost identical functions with slightly different names or the names of the variables (feaf node data)
- Dropout and Weight Decay not implemented for overfit prevention

In later section, we will try to reduce this overfit by preprocessing the data with taking into account these assumptions.

3.1.2 Testing

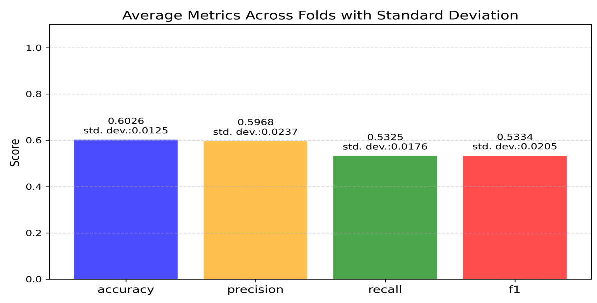


Figure 3.2: Average testing metrics across folds.

For testing visualization purposes, we have also chosen accuracy, precision, f1 **metrics average across the folds** heatmap (respectively in fig. 3.3) because confusion matrix for all the classes is hard to analyze with hundred of classes. What interested us, is that the model started to perfoms worse when the size of class in test set started to be lower than 4. One could say that size of 3 is too small to take it as relevant information, but as the heat map is average across the folds, we can interpret it as average accuracy of 5*3 = 15 tests. Going closer to size of test set = 2 the blue spots indicating low performence start to appear even more. The whole heatmap is available to see in the attachments. Also the best performances were reached by the functions with names of standard algorithms like quicksort, mergesort, dfs, backtrack or gcd.

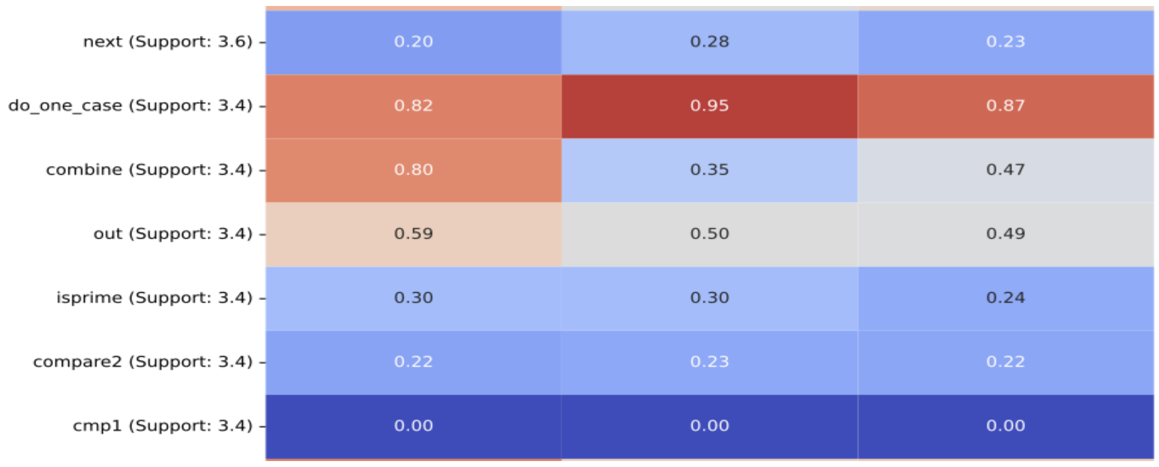


Figure 3.3: Snippet from heatmap of names as classes.

To evaluate the model's performance on different subsets of the data, we divided the dataset into bins based on specific features, such as the number of nodes, AST depth, or token counts. For each bin, we calculated the average accuracy and visualized the results using bar plots. This allowed us to compare the model's performance across different feature ranges and identify any variations. The overall model accuracy, represented by the red dashed line, serves as a reference for interpreting the results.

3.2 Extensive Data Preprocessing

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] Leandro T. C. Melo, Rodrigo G. Ribeiro, Breno C. F. Guimarães, and Fernando Magno Quintão Pereira. Type inference for c: Applications to the static analysis of incomplete programs. *ACM Trans. Program. Lang. Syst.*, 42(3), November 2020.

[6] 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.

[7] 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.

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