Article

Not peer-reviewed version

Structural Syntax Network for Code Classification

Mike Shah [*] , Rodolfo Patel , Ango Terry

Posted Date: 12 December 2023

doi: 10.20944/preprints202312.0805.v1

Keywords: code analysis; syntax-based classification; multi-language code parsing; cross-linguistic code analysis

Article

Structural Syntax Network for Code Classification

Mike Shah *, Rodolfo Patel and Ango Terry

Briar Cliff University
* Correspondence: mshah@briarcliff.edu

Abstract: The field of program classification, a critical aspect of software engineering, facilitates understanding and categorization of code across various applications, including anomaly detection, and code quality assessment. The advancement of cross-language program classification opens up avenues for efficient code translation among different programming languages, significantly aiding developers in rapid coding and reducing development cycles. Existing research primarily focuses on semantic code analysis, with limited emphasis on cross-linguistic challenges. To address this, we introduce an innovative neural network model, CodeSemanticsNN, which leverages a refined Syntax Structure (SS) approach. This model consists of two integral mechanisms: first, it adopts a novel SS representation that combines sequential and graph-based SS structures, enhancing semantic feature capture. Second, it employs a 'unified vocabulary' strategy to bridge the gap between diverse programming languages, facilitating efficient cross-language classification. Additionally, we have compiled a comprehensive dataset of 20,000 files spanning five programming languages, serving as a benchmark for cross-language classification. Our experiments on this dataset demonstrate that CodeSemanticsNN surpasses existing models in four key metrics: Precision, Recall, F1-score, and Accuracy.

Keywords: code analysis; syntax-based classification; multi-language code parsing; cross-linguistic code analysis

1. Introduction

The field of program classification, pivotal in automating the categorization of software based on functionality and semantic content, has emerged as a cornerstone in understanding and manipulating source code. This domain has seen extensive applications in tasks like detecting code clones [1–4], identifying defects [7–10], and aiding in code retrieval [15,16]. Traditionally reliant on manual effort, which is labor-intensive and prone to errors, program classification has shifted towards automation, leveraging the structured nature of source codes for more efficient processing.

Cross-language program classification extends this concept to the realm of multi-language environments. It involves categorizing code from diverse programming languages based on their structural and semantic properties. Consider, for instance, the sorting algorithms: *"quick sort"* in C++ and *"bubble sort"* in Python. Despite their differing implementations, both share a sorting functionality and thus, should be grouped under the same category. This cross-language approach not only streamlines the process of implementing similar functionalities in different languages but also fosters an environment conducive to code reuse across languages, thereby expediting development and reducing time-to-market.

The challenge, however, lies in effectively and precisely capturing the semantic essence of code across varied programming languages, each with its unique syntax and coding paradigms. Prior research has approached this from several angles. Harer et al. [17], for instance, utilized token-based methods to embed vectors and process them through neural networks for feature extraction. However, tokens capture only lexical information and fail to reflect the deeper structural and semantic nuances of code. Ben et al. [18] employed Intermediate Representations (IRs) to create XFG graphs, subsequently analyzed using neural networks like GNN [19] and RNN [20]. While effective in semantic extraction, the requirement for complete code compilation limits the applicability of IRs, especially for partial code snippets. The Abstract Syntax Tree (AST) based approaches proposed by Azcona et al. [21] and Mou et al. [22] offered insights into feature extraction from traversal or tree structures of ASTs, yet they predominantly focused on single-language semantics.

Addressing the gaps in cross-language program classification, we propose a novel neural network model, the **Syntax Structure (SS)**. SS is designed to extract semantic features using a dual-pronged approach: employing self-attention mechanisms combined with Bi-LSTM for a flattened AST sequence, capturing the overarching logical structure of the code, and utilizing a Graph Convolutional Neural (GCN) [23] network for analyzing graph-like AST structures at a local level. By merging these sequence and graph features, SS enhances the dimensional representation of structural and semantic code characteristics. To further bolster cross-language semantic learning, we introduce a 'unified vocabulary' strategy, standardizing embedded mapping across different programming languages and easing the neural network's learning process. We also curated a dataset from Leetcode comprising five distinct programming languages, ideal for benchmarking in cross-language classification. Our experimental analysis on two datasets indicates that SS outperforms leading models like CodeBERT and Infercode in terms of Recall, Precision, F1-score, and Accuracy. Additionally, we conducted ablation studies to assess the impact of our unified vocabulary and AST feature fusion on model performance.

In summary, our primary contributions are as follows:

- Introduction of the Syntax Structure (SS) network, employing two sub-networks (SAST and GAST) for unified AST representation learning. SAST extracts global syntactic features from AST path sequences, while GAST captures local semantic features within the AST tree. This comprehensive approach ensures effective learning of code semantics.
- Development of a unified vocabulary mechanism to minimize language disparities, allowing for efficient embedded vector mapping and training in cross-language program classification.
- Performance evaluation on two datasets, demonstrating SS's superiority over existing baselines in key metrics like Recall, Precision, F1-score, and Accuracy.
- Compilation of a benchmark dataset for cross-language program classification, encompassing five programming languages (C, C++, Java, Python, and JavaScript), with 50 problems and 20,000 solution files that are semantically akin.

The remainder of this paper is organized as follows: Section 2 delves into the Syntax Structure (SS) network. Sections 3 and 4 discuss the experimental setup and results. Section 5 reviews related work. Section 6 presents a discussion of our findings. Finally, Section 7 concludes the paper and outlines future research directions.

2. Related Work

The concept of program classification has evolved as a key abstraction layer in code analysis, serving as a cornerstone for various source code comprehension tasks in software engineering. Its applications extend to diverse areas such as code clone detection [2,4,24,25], bug resolution [26], code quality assessment [27], defect identification [7], and program understanding [28]. The genesis of program classification traces back several decades, with initial studies [4,29,33,34] utilizing basic features like tokens, component names, and code metrics for classification. These early approaches, though limited in performance due to their reliance on superficial code features and rule-based classification, laid the groundwork for future advancements.

The integration of machine learning has revolutionized program classification. Techniques such as Support Vector Machine (SVM) [35] have been employed for classifying code tokens [36], and various models like SVM, Decision Tree [37], and Bayesian Network [38] have been used to classify code based on token sequences [39]. The incorporation of deep learning has further propelled this field. Mou et al. [22] introduced a tree-based convolutional network, adept at capturing the structural intricacies of source code. Zhang et al. [40] innovatively divided the AST into sub-trees and employed Bi-RNN models for vector representation, achieving remarkable success in code classification. Barchi et al. [41] demonstrated the efficacy of Convolutional Neural Networks (CNN) [42] in analyzing program source code, establishing CNN as a superior choice in terms of accuracy and learning efficiency.

The advent of deep neural network technology has brought transformative changes to software engineering, particularly in code representation learning. Various forms of code representation, such as tokens for lexical information, Abstract Syntax Trees (AST) for structural and grammatical insights, Data Control Flow (DFG), and Control Flow Graph (CFG) for understanding code flows, have been utilized alongside neural networks to uncover deeper code semantics.

Harer et al. [17] leveraged the Word2Vec [43] tool to create initial vectors for C/C++ tokens, applying the TextCNN [44] model for feature learning in software vulnerability detection. Azcona et al. [21] embedded Python code tokens into vectors for student performance analysis. Mou et al. [22] developed a TBCNN model based on AST, using a sub-tree kernel and dynamic pooling for structural extraction. Alon et al. [28] introduced code2vec, applying attention mechanisms to AST paths for syntactic information extraction. Wang et al. [45] created the Flow-Augmented AST (FA-AST) to represent source code and applied GNNs [19] for feature extraction. Ben-Nun et al. [18] explored code semantics learning based on IRs, transforming them into context flow graphs (XFG) and applying neural networks for feature learning. Wei et al. [46] used type dependency graphs, combined with GNNs, to extract features from IRs.

Cross-language code learning, a nascent yet promising field, draws parallels with machine translation in NLP [47,48]. Nguyen et al. [49] introduced semSMT, a tool for migrating Java to C# at lexical, syntactic, and semantic levels. Bui et al. [50] proposed a bilateral dependency neural network leveraging Siamese network architectures for feature learning across languages. The MISIM model by Ye et al. [51] employed the Context-Aware Semantic Structure (CASS) for language-independent representation learning. Bui et al.'s Infercode [52] adopted self-supervised learning, training on AST contexts across multiple languages. Peng et al.'s TPTrans [53] integrated code tree features into transformer structures, showing effectiveness in code summarization tasks.

3. The Proposed Approach

Herein, we introduce the Syntax Structure (SS) model, a sophisticated neural network architecture engineered for the nuanced task of cross-language program classification. This model is a culmination of cutting-edge techniques in programming language processing, adapted to understand and categorize diverse coding languages.

3.1. Overall Structure

SS begins by ingesting source code from various programming languages, transforming them into Abstract Syntax Trees (ASTs). Leveraging a unique unified vocabulary, the model then applies both path and graph embeddings to these ASTs. Distinct sub-networks, designed to capture specific code features from these embeddings, process the path-embedded and graph-embedded vectors separately. The model culminates its process by fusing these separately learned features for the final classification task.

This innovative model is structured into five core sections: Unified Vocabulary, Sequence-based AST Network (SAST), Graph-based AST Network (GAST), Unified AST Feature Fusion, and Cross-language Program Classification. Each section contributes uniquely to the model's overall functionality.

3.2. Unified Vocabulary

The Unified Vocabulary mechanism is a pivotal aspect of the SS model, addressing the challenge posed by varying textual representations and coding rules across different programming languages. By normalizing AST node names, such as harmonizing different terms like "program," "translation unit," and "module" into a common label "unit," the model significantly reduces linguistic discrepancies.

This standardization is crucial for the model to learn and classify features consistently across various programming languages.

3.3. Sequence-based AST Network (SAST)

The SAST component is integral to the SS model, focusing on extracting the syntactic structure from the code. It begins with a pre-order traversal of the unified AST, producing a path sequence that flattens the tree structure. This sequence contains comprehensive information about the global structure and syntax of the source code. To interpret the intricate relationships within these sequences, the model employs self-attention mechanisms, as formulated:

$$Attention(Q, K, V) = SoftMax(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

Here, Q, K, and V represent matrices initialized from the embedded path sequence vector. The self-attention mechanism, part of the transformer architecture, excels in extracting internal relationships and mitigating long-distance dependency issues.

Furthermore, to capture the context of the code, which is crucial for understanding its intent and logic, the model integrates Bi-directional Long Short-Term Memory (Bi-LSTM). This component processes the code information bidirectionally, ensuring a comprehensive understanding of the context. The Bi-LSTM computation is expressed as follows:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2}$$
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{4}$$
$$\tilde{c}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_c) \tag{5}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{6}$$
$$h_t = o_t \odot tanh(c_t) \tag{7}$$

The combination of self-attention and Bi-LSTM enables the SAST to effectively capture both the syntactic structure and the contextual logic of the source code.

3.4. Graph-based AST Network (GAST)

Complementing the SAST, the GAST focuses on learning the local tree-like structural features of the AST. This network treats the AST as a unique graph, applying convolution operations to learn the semantic features inherent in the code's structure. By aggregating information from neighbor nodes, GAST adeptly captures the local structural and semantic nuances, as elucidated:

$$\widetilde{A} = A + I \tag{8}$$
$$H_i^{(l+1)} = \sigma(\sum_{j \in N} D^{-\frac{1}{2}} \widetilde{A} D^{-\frac{1}{2}} H_j^{(l)} W^{(l)}) \tag{9}$$

The GAST thereby contributes significantly to the model's understanding of the nuanced semantics and logic of code, irrespective of the programming language.

3.5. Unified AST Feature Fusion

To synthesize the insights garnered by SAST and GAST, the SS model employs a feature fusion mechanism. This process involves concatenating the respective feature vectors from each network to form a unified representation, embody

doi:10.20944/preprints202312.0805.v1

ing both the global structural and local semantic aspects of the code:

$$h_{code} = concat(h_{SAST}, h_{GAST}) \tag{10}$$

Through this fusion, the model achieves a holistic understanding of the code's characteristics, setting the stage for effective cross-language classification.

3.6. Cross-language Program Classification

In its final stage, SS utilizes the comprehensive feature vector for classifying programs across different languages. A fully connected layer transforms these features into a format suitable for classification, with the probabilities for each category derived through a SoftMax layer. The model utilizes Cross Entropy as its loss function, optimizing it with the Adam optimizer to ensure accurate and efficient classification:

$$J = -\sum_{i=1}^{k} y_i log(p_i) \tag{11}$$

In conclusion, the Syntax Structure (SS) model represents a sophisticated and holistic approach to cross-language program classification, harnessing the power of modern neural network techniques to understand and categorize diverse programming languages.

4. Experiments

This section elaborates on our investigative approach by presenting research questions (RQs), detailing the experimental datasets, comparing baseline models, specifying experimental settings, and outlining common evaluation metrics.

4.1. Targets

Our investigation focuses on the efficacy of the Syntax Structure (SS) model, particularly its novel unified vocabulary and AST feature fusion techniques. The following research questions guide our exploration:

RQ1: What is the comparative performance of SS against established baselines?

RQ1 evaluates the SS model's effectiveness relative to other leading methodologies in code representation learning for cross-language program classification. We compare SS with several state-of-the-art baselines, as detailed in Section 3.3.

RQ2: How does the unified vocabulary component influence SS's performance?

RQ2 aims to assess the impact of the unified vocabulary feature in SS. This question is addressed through ablation studies, analyzing how the inclusion or exclusion of this component affects overall model performance.

RQ3: What is the effect of the unified AST feature fusion on SS's efficacy?

RQ3 examines the role of the unified AST feature fusion in SS. Given that the sub-networks SAST and GAST extract global syntactic and local semantic features respectively, we investigate how their combined effect enhances model performance.

RQ4: How do various parameter configurations impact SS's performance?

RQ4 explores the influence of different parameter settings within SS. Key parameters, such as the path length in AST traversal and the number of GCN layers, are adjusted to identify the most effective configuration.

4.2. Datasets

We utilize two distinct datasets for model evaluation:

- `JC`: Sourced from Bi-TBCNN by Bui et al.[50], this dataset, named JC, comprises 5822 Java files and 7019 C++ files across 10 program categories from GitHub. It is partitioned into training, validation, and testing sets, as outlined in Table 1.
- `Leetcode`: A comprehensive dataset from Leetcode featuring 20000 solutions in five languages (C, C++, Java, Python, and JavaScript) across 50 categories. We employ NICAD for deduplication and anonymize function names to maintain consistency. This dataset also follows a training-validation-testing split, with the distribution detailed in Table 1.

Table 1. Dataset Distribution for JC and Leetcode.

	JC		Leetcode				
	Java	C++	C	C++	Java	Python	JavaScript
Training	3498	4215	331	3428	5051	2633	557
Validation	1162	1402	110	1143	1684	878	185
Testing	1162	1402	110	1143	1684	878	185

4.3. Baselines

In our study, we compare the effectiveness of the Syntax Structure (SS) model against several leading baselines in the realm of cross-language code learning:

- `Bi-TBCNN` [50]: This approach utilizes tree-based neural networks to decipher cross-language programming paradigms. It features a dual-neural network framework, where each network encodes and interprets the grammatical and semantic aspects of language codes, thus facilitating cross-language program classification.
- `CodeBERT` [54]: A BERT [55] adaptation, CodeBERT employs Replaced Token Detection (RTD) for its initial training. This model utilizes deep bi-transformer components to generate code features that integrate contextual information, effectively capturing the nuances of input sequences. Given its training on six programming languages, it stands as a robust tool for cross-language classification.
- `Infercode` [52]: Employing self-supervised learning principles from natural language processing, this model trains code representations by predicting the contextual subtrees of ASTs. Trained on a multitude of programming languages, Infercode is adept at extracting and classifying features unique to different languages, making it suitable for cross-language program classification.

4.4. Setting

Our implementation of the SS model uses Pytorch. We conduct all experiments on a robust setup featuring a 10-core 3.70GHz Intel(R) Core(TM) i9-10900X CPU and an NVIDIA GeForce RTX 3090 GPU. To maintain the integrity of our comparative analysis, we utilize a consistent training dataset across all models. Our parameter tuning, performed on the validation set, includes setting the SS model's epoch to 5, with a batch size of 64. We utilize the Adam optimizer [56], setting the learning rate at 0.001. For the Sequence-based AST Network (SAST) component, we standardize the path sequence length to 700 for the Dataset JC and 200 for the Dataset Leetcode. We set the path embedding dimension to 200, the Bi-LSTM unit count to 64, and the Bi-LSTM layer count to 2, with a dropout rate of 0.5 for Bi-LSTM and 0.2 for self-attention. The self-attention layer utilizes 4 heads. The Graph-based AST Network (GAST) uses an adjacency matrix size of [400, 400], and we implement two layers of GCN.

4.5. Evaluation Metrics

To evaluate our multi-classification model, we employ well-established metrics: Precision, Recall, F1-score, and Accuracy. These metrics are widely used in similar studies [40] and provide a comprehensive assessment of model performance. The mathematical definitions of these metrics are as follows:

$$Precision = \frac{\overline{TP}}{\overline{TP} + \overline{FP}} \tag{12}$$

$$Recall = \frac{\overline{TP}}{\overline{TP} + \overline{FN}} \tag{13}$$

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{14}$$

$$Accuracy = \frac{\overline{TP} + \overline{TN}}{\overline{TP} + \overline{FP} + \overline{TN} + \overline{FN}} \tag{15}$$

In these formulas, $\overline{TP}$ (True Positive) represents the weighted average of correctly predicted positive samples for each class; $\overline{FN}$ (False Negative) indicates the weighted average of samples wrongly predicted as negative per class; $\overline{FP}$ (False Positive) is the weighted average of incorrectly predicted positive samples per class; $\overline{TN}$ (True Negative) denotes the weighted average of correctly predicted negative samples per class.

4.6. Results and Analysis

This section delves into the experimental outcomes, addressing the four research questions delineated in Section 4.1.

4.6.1. Model Performance

To address this, we compare the Syntax Structure (SS) model with the algorithms mentioned in Section 3.3. The comparative results on both datasets are encapsulated in Table 2 and Table 3. From Table 2, it is evident that SS excels with a Recall of 0.9611, Precision of 0.9631, F1-score of 0.9617, and Accuracy of 0.9626, thereby surpassing other baselines. Similarly, SS demonstrates superior performance on the Leetcode dataset, reinforcing its efficacy in cross-language program classification.

Table 2. Comparative experiment on Dataset JC.

Model	Recall	Precision	F1-score	Accuracy
CodeBERT	0.9078	0.9177	0.9090	0.9005
Infercode	0.8317	0.8468	0.8325	0.8343
SS	**0.9611**	**0.9631**	**0.9617**	**0.9626**

Table 3. Comparative experiment on Dataset Leetcode.

Model	Recall	Precision	F1-score	Accuracy
CodeBERT	0.6147	0.6348	0.6174	0.6245
Infercode	0.5696	0.5819	0.5762	0.5807
SS	**0.7958**	**0.8025**	**0.7965**	**0.7964**

The SS model, integrating global and local code structure semantics and leveraging a unified vocabulary, has demonstrated a notable improvement over other baselines. This is especially significant in its ability to bridge the gap across various programming languages, thus enhancing its cross-language classification performance.

The experiments conducted on datasets of varying sizes and language diversities not only underscore the efficiency of the SS model but also its robust generalization capabilities. The

Syntax Structure (SS) model significantly outperforms other state-of-the-art baselines across all key metrics, including Precision, Recall, F1-score, and Accuracy, thereby validating its effectiveness in cross-language program classification.

4.6.2. The Impact of the Unified Vocabulary

In response, we examine the influence of the unified vocabulary mechanism, integral to SS, through an ablation study. The results, presented in Table 4, highlight the performance differences when the unified vocabulary is applied versus when it is not.

Table 4. Ablation experiment results of the unified vocabulary on Dataset JC and Leetcode.

Dataset	Model	Recall	Precision	F1-score	Accuracy
JC	SS-V	0.8802	0.8868	0.8816	0.8856
	SS	**0.9125**	**0.9254**	**0.9142**	**0.9142**
	GAST-V	0.9467	0.9479	0.9469	0.9478
	GAST	**0.9504**	**0.9516**	**0.9511**	**0.9508**
	SS-V	0.9524	0.9526	0.9509	0.9516
	SS	**0.9611**	**0.9631**	**0.9617**	**0.9626**
Leetcode	SS-V	0.6553	0.6894	0.6540	0.6554
	SS	**0.6718**	**0.7020**	**0.6707**	**0.6721**
	GAST-V	0.7744	0.7793	0.7749	0.7749
	GAST	**0.7892**	**0.7956**	**0.7887**	**0.7892**
	SS-V	0.7893	0.7970	0.7904	0.7882
	SS	**0.7958**	**0.8025**	**0.7965**	**0.7964**

The data in Table 4 show that utilizing the unified vocabulary enhances performance across all metrics on both the JC and Leetcode datasets. This indicates that the unified vocabulary successfully minimizes feature discrepancies between different programming languages, thereby boosting cross-language classification performance.

The unified vocabulary's most significant impact is observed in the Sequence-based AST Network (SAST). Given that SAST relies heavily on initial vector embedding, which is directly influenced by vocabulary mapping, this finding aligns with expectations. The Graph-based AST Network (GAST), while also benefiting from the unified vocabulary, shows a comparatively lesser impact due to its adjacency matrix construction based on AST structure rather than vocabulary.

In summary, the unified vocabulary mechanism plays a crucial role in enhancing the performance of the SS model, particularly in cross-language contexts, affirming its effectiveness in bridging language feature gaps. The unified vocabulary mechanism significantly enhances the performance of the Syntax Structure (SS) model, particularly evident in the Sequence-based AST Network, validating its role in reducing feature disparities across different programming languages.

4.6.3. The Impact of the AST Feature Fusion

To elucidate the impact of the unified AST feature fusion on the Syntax Structure (SS) model, we present our findings. These results indicate a significant enhancement in model performance after feature fusion. On Dataset JC, we observe improvements ranging from 1.06% to 4.86%, while on Dataset Leetcode, the enhancements range from 0.66% to 12.58%. This clearly demonstrates that the unified AST feature fusion substantially boosts the effectiveness of the SS model. Notably, the GAST network exhibits superior performance over the SAST network on both datasets. This suggests that features derived from the graph-like AST structure are more informative than those from the flattened AST sequence. The flattened sequence, lacking branch statement structures, only provides an overall code process, whereas the graph-like AST's adjacency matrix captures intricate relationships between nodes, crucial for understanding specific code structures like loops and conditionals. The unified AST feature fusion mechanism combines the strengths of both global and local code feature extraction. By fusing the hidden vectors from SAST and GAST, the SS model effectively leverages comprehensive code semantics from multiple perspectives, leading to enhanced performance. The unified AST feature fusion in the SS model, integrating global structural and local semantic information, significantly improves model performance, highlighting the importance of a comprehensive code representation approach.

4.6.4. The Impact of Parameter Settings

Investigating the impact of key parameters within the Syntax Structure (SS) model, specifically the length of the path sequence in SAST and the layers of the GCN in GAST, is essential. These parameters critically influence the model's effectiveness. The path sequence length governs the extent of code content processed by the neural network, and the GCN layers (hops) determine the scope of neighbor node information aggregation. To address this, we conduct experiments with varying parameter settings.

Path Sequence Length: We experiment with different path sequence lengths (ranging from 100 to 1000) to evaluate their impact on the model's performance. We demonstrate that SS reaches its peak performance at a sequence length of 700 for Dataset JC and 200 for Dataset Leetcode across all metrics (Recall, Precision, F1-score, and Accuracy). Longer sequences do not enhance performance and add to the model's complexity. Additionally, Bi-LSTM's inefficiency with extended sequences could contribute to the diminished model efficacy. Conversely, shorter sequences also degrade performance due to the loss of essential code information.

AST Path Length Statistics: As shown in Table 5, a significant proportion (80%) of AST path sequences fall within certain lengths on both datasets (726 for Dataset JC and 221 for Dataset Leetcode). This insight guides the optimal setting of sequence input lengths, balancing the inclusion of critical code content while avoiding excessive padding or information loss.

Table 5. Statistical distribution of AST path sequence length on Dataset JC and Leetcode.

Dataset	Mean	Median	70%	80%	90%
JC	576	354	502	726	1498
Leetcode	165	144	189	221	279

GCN Layer Impact: The model's efficacy with different GCN layers (1, 2, 3) is assessed, with optimal performance observed at two layers. This setting provides a balanced neighbor information range, capturing specific code structures effectively, whereas a greater number of layers might overextend the neighbor range and risk overfitting.

From these observations, we infer that the choice of path length should be made after careful analysis of dataset-specific AST path lengths, selecting the most suitable value. Moreover, setting the GCN layer count to 2 appears to enhance model performance, striking a balance between capturing local semantic information and maintaining model simplicity. Optimal path sequence length and GCN layer settings are crucial for the performance of the Syntax Structure (SS) model. Statistical analysis of AST paths guides the selection of appropriate sequence lengths, while a two-layer GCN configuration is found to be most effective.

5. Conclusions

This research presents an advanced approach to enhancing cross-language code comprehension, introducing the Syntax Structure (SS) neural network model. This innovative framework, initially conceptualized as the Unified Abstract Syntax Tree, is specifically designed for the intricate task of cross-language program classification. The SS model ingeniously integrates two distinct sub-networks: the Sequence-based AST Network (SAST) and the Graph-based AST Network. The SAST component is adept at extracting comprehensive syntactic features from the AST path sequence, providing a global perspective of the code structure. In contrast, GAST delves into the local semantics embedded within the AST tree, capturing the intricate nuances of code semantics at a granular level. This duality of the SS model allows for a holistic understanding of code, merging both global syntactic and local semantic features through its innovative unified AST feature fusion mechanism. This fusion not only bridges the gap between the two distinct code feature perspectives but also enriches the overall semantic understanding of the code. Another cornerstone of the SS model is the unified vocabulary mechanism. This unique feature standardizes terminology across various programming languages, significantly reducing disparities in coding syntax and semantics. This uniformity in language representation is pivotal in enhancing the model's ability to accurately classify and interpret code written in different programming languages. Empirical evaluations, conducted on both publicly available datasets and our custom-compiled dataset, underscore the superior performance of the SS model compared to existing state-of-the-art baselines. The SS model demonstrates an exceptional ability to differentiate between diverse program types, marking a significant leap in the field of AI-driven software engineering. Looking ahead, the future trajectory of our research aims to integrate source code analysis with natural language processing. This ambitious endeavor seeks to further refine program classification techniques, incorporating a richer, more nuanced understanding of programming languages by harnessing the descriptive power of natural language text. This fusion of code and text information promises to unlock new frontiers in the realm of AI-assisted software development and analysis.

References

1. Baker, B.S. A program for identifying duplicated code. *Computing Science and Statistics* **1993**, pp. 49–49.
2. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* **2007**, *33*, 577–591.
3. Fei, H.; Ren, Y.; Ji, D. Retrofitting Structure-aware Transformer Language Model for End Tasks. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, 2020, pp. 2151–2161.
4. Börstler, J. Feature-oriented classification for software reuse. Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering. Knowledge Systems Institute, 1995, Vol. 95, pp. 22–24.
5. Fei, H.; Ren, Y.; Ji, D. Boundaries and edges rethinking: An end-to-end neural model for overlapping entity relation extraction. *Information Processing & Management* **2020**, *57*, 102311.
6. Li, J.; Fei, H.; Liu, J.; Wu, S.; Zhang, M.; Teng, C.; Ji, D.; Li, F. Unified Named Entity Recognition as Word-Word Relation Classification. Proceedings of the AAAI Conference on Artificial Intelligence, 2022, pp. 10965–10973.
7. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. Proceedings of the 38th International Conference on Software Engineering. ACM, 2016, pp. 297–308.
8. Li, J.; Xu, K.; Li, F.; Fei, H.; Ren, Y.; Ji, D. MRN: A Locally and Globally Mention-Based Reasoning Network for Document-Level Relation Extraction. Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, 2021, pp. 1359–1370.
9. Fei, H.; Wu, S.; Ren, Y.; Zhang, M. Matching Structure for Dual Learning. Proceedings of the International Conference on Machine Learning, ICML, 2022, pp. 6373–6391.
10. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software defect prediction via convolutional neural network. Proceedings of the 2017 International Conference on Software Quality, Reliability and Security. IEEE, 2017, pp. 318–328.
11. Wu, S.; Fei, H.; Li, F.; Zhang, M.; Liu, Y.; Teng, C.; Ji, D. Mastering the Explicit Opinion-Role Interaction: Syntax-Aided Neural Transition System for Unified Opinion Role Labeling. Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence, 2022, pp. 11513–11521.
12. Shi, W.; Li, F.; Li, J.; Fei, H.; Ji, D. Effective Token Graph Modeling using a Novel Labeling Strategy for Structured Sentiment Analysis. Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2022, pp. 4232–4241.
13. Fei, H.; Zhang, Y.; Ren, Y.; Ji, D. Latent Emotion Memory for Multi-Label Emotion Classification. Proceedings of the AAAI Conference on Artificial Intelligence, 2020, pp. 7692–7699.

14. Wang, F.; Li, F.; Fei, H.; Li, J.; Wu, S.; Su, F.; Shi, W.; Ji, D.; Cai, B. Entity-centered Cross-document Relation Extraction. Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 9871–9881.
15. Kim, K.; Kim, D.; Bissyandé, T.F.; Choi, E.; Li, L.; Klein, J.; Traon, Y.L. FaCoY: a code-to-code search engine. Proceedings of the 40th International Conference on Software Engineering. ACM, 2018, pp. 946–957.
16. Fei, H.; Wu, S.; Ren, Y.; Li, F.; Ji, D. Better Combine Them Together! Integrating Syntactic Constituency and Dependency Representations for Semantic Role Labeling. Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, 2021, pp. 549–559.
17. Harer, J.A.; Kim, L.Y.; Russell, R.L.; Ozdemir, O.; Kosta, L.R.; Rangamani, A.; Hamilton, L.H.; Centeno, G.I.; Key, J.R.; Ellingwood, P.M.; others. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497* **2018**.
18. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural code comprehension: A learnable representation of code semantics. Advances in Neural Information Processing Systems. Curran Associates, Inc, 2018, Vol. 31, pp. 3585–3597.
19. Scarselli, F.; Gori, M.; Tsoi, A.C.; Hagenbuchner, M.; Monfardini, G. The graph neural network model. *IEEE transactions on neural networks* **2008**, *20*, 61–80.
20. Elman, J.L. Finding structure in time. *Cognitive science* **1990**, *14*, 179–211.
21. Azcona, D.; Arora, P.; Hsiao, I.H.; Smeaton, A. user2code2vec: Embeddings for profiling students based on distributional representations of source code. Proceedings of the 9th International Conference on Learning Analytics & Knowledge. ACM, 2019, pp. 86–95.
22. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. Proceedings of the 30th AAAI Conference on Artificial Intelligence. ACM, 2016.
23. Kipf, T.N.; Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* **2016**.
24. Wu, S.; Fei, H.; Ren, Y.; Ji, D.; Li, J. Learn from Syntax: Improving Pair-wise Aspect and Opinion Terms Extraction with Rich Syntactic Knowledge. Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, 2021, pp. 3957–3963.
25. Fei, H.; Li, F.; Li, B.; Ji, D. Encoder-Decoder Based Unified Semantic Role Labeling with Label-Aware Syntax. Proceedings of the AAAI Conference on Artificial Intelligence, 2021, pp. 12794–12802.
26. Peters, F.; Tun, T.T.; Yu, Y.; Nuseibeh, B. Text filtering and ranking for security bug report prediction. *IEEE Transactions on Software Engineering* **2017**, *45*, 615–631.
27. Fontana, F.A.; Zanoni, M. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* **2017**, *128*, 43–58.
28. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* **2019**, *3*, 1–29.
29. Clark, K.L.; Darlington, J. Algorithm classification through synthesis. *The computer journal* **1980**, *23*, 61–65.
30. Fei, H.; Wu, S.; Li, J.; Li, B.; Li, F.; Qin, L.; Zhang, M.; Zhang, M.; Chua, T.S. LasUIE: Unifying Information Extraction with Latent Adaptive Structure-aware Generative Language Model. Proceedings of the Advances in Neural Information Processing Systems, NeurIPS 2022, 2022, pp. 15460–15475.
31. Wu, S.; Fei, H.; Ji, W.; Chua, T.S. Cross2StrA: Unpaired Cross-lingual Image Captioning with Cross-lingual Cross-modal Structure-pivoted Alignment. Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2023, pp. 2593–2608.
32. Fei, H.; Ren, Y.; Zhang, Y.; Ji, D.; Liang, X. Enriching contextualized language model from knowledge graph for biomedical information extraction. *Briefings in Bioinformatics* **2021**, *22*.
33. Jiang, L.; Su, Z. Automatic mining of functionally equivalent code fragments via random testing. Proceedings of the 18th International Symposium on Software Testing and Analysis. ACM, 2009, pp. 81–92.
34. Taherkhani, A.; Korhonen, A.; Malmi, L. Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms. *The Computer Journal* **2011**, *54*, 1049–1066.
35. Drucker, H.; Burges, C.J.; Kaufman, L.; Smola, A.; Vapnik, V.; others. Support vector regression machines. *Advances in neural information processing systems* **1997**, *9*, 155–161.

36. Ugurel, S.; Krovetz, R.; Giles, C.L. What's the code? automatic classification of source code archives. Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2002, pp. 632–638.
37. Quinlan, J.R. Induction of decision trees. *Machine learning* **1986**, *1*, 81–106.
38. Friedman, N.; Geiger, D.; Goldszmidt, M. Bayesian network classifiers. *Machine learning* **1997**, *29*, 131–163.
39. Ma, Y.; Fakhoury, S.; Christensen, M.; Arnaoudova, V.; Zogaan, W.; Mirakhorli, M. Automatic classification of software artifacts in open-source applications. Proceedings of the 15th International Conference on Mining Software Repositories. ACM, 2018, pp. 414–425.
40. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. Proceedings of the 41st International Conference on Software Engineering. IEEE, 2019, pp. 783–794.
41. Barchi, F.; Parisi, E.; Urgese, G.; Ficarra, E.; Acquaviva, A. Exploration of Convolutional Neural Network models for source code classification. *Engineering Applications of Artificial Intelligence* **2021**, *97*, 104075.
42. Kalchbrenner, N.; Grefenstette, E.; Blunsom, P. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188* **2014**.
43. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* **2013**.
44. Zhang, Y.; Wallace, B. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820* **2015**.
45. Wang, W.; Li, G.; Ma, B.; Xia, X.; Jin, Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering. IEEE, 2020, pp. 261–271.
46. Wei, J.; Goyal, M.; Durrett, G.; Dillig, I. Lambdanet: Probabilistic type inference using graph neural networks. Proceedings of the 8th International Conference on Learning Representations. OpenReview, 2020.
47. Huang, J.T.; Li, J.; Yu, D.; Deng, L.; Gong, Y. Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers. Proceedings of the 2013 International Conference on Acoustics, Speech and Signal Processing. IEEE, 2013, pp. 7304–7308.
48. Conneau, A.; Lample, G. Cross-lingual language model pretraining. Advances in Neural Information Processing Systems. Curran Associates, Inc., 2019, Vol. 32, pp. 7059–7069.
49. Nguyen, A.T.; Nguyen, T.T.; Nguyen, T.N. Migrating code with statistical machine translation. Proceedings of the 36th International Conference on Software Engineering Companion. ACM, 2014, pp. 544–547.
50. Bui, N.D.; Yu, Y.; Jiang, L. Bilateral dependency neural networks for cross-language algorithm classification. Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering. IEEE, 2019, pp. 422–433.
51. Ye, F.; Zhou, S.; Venkat, A.; Marucs, R.; Tatbul, N.; Tithi, J.J.; Petersen, P.; Mattson, T.; Kraska, T.; Dubey, P.; others. Misim: An end-to-end neural code similarity system. *arXiv preprint arXiv:2006.05265* **2020**.
52. Bui, N.D.; Yu, Y.; Jiang, L. InferCode: Self-supervised learning of code representations by predicting subtrees. Proceedings of the 43rd International Conference on Software Engineering. IEEE, 2021, pp. 1186–1197.
53. Peng, H.; Li, G.; Wang, W.; Zhao, Y.; Jin, Z. Integrating tree path in transformer for code representation. Proceedings of the 35th Conference on Neural Information Processing Systems. Curran Associates, Inc., 2021.
54. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; others. CodeBERT: A pre-trained model for programming and natural languages. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings. ACL, 2020, pp. 1536–1547.
55. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. ACL, 2019, pp. 29–35.
56. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. Proceedings of the 3rd International Conference on Learning Representations. OpenReview, 2015.