SANN: Programming Code Representation Using Attention Neural Network with Optimized Subtree Extraction

Muntasir Hoq
North Carolina State University
Raleigh, NC, USA
mhoq@ncsu.edu

Sushanth Reddy Chilla
North Carolina State University
Raleigh, NC, USA
schilla@ncsu.edu

Melika Ahmadi Ranjbar
North Carolina State University
Raleigh, NC, USA
mahmadi@ncsu.edu

Peter Brusilovsky
University of Pittsburgh
Pittsburgh, PA, USA
peterb@pitt.edu

Bita Akram
North Carolina State University
Raleigh, NC, USA
bakram@ncsu.edu

ABSTRACT

Automated analysis of programming data using code representation methods offers valuable services for programmers, from code completion to clone detection to bug detection. Recent studies show the effectiveness of Abstract Syntax Trees (AST), pre-trained Transformer-based models, and graph-based embeddings in programming code representation. However, pre-trained large language models lack interpretability, while other embedding-based approaches struggle with extracting important information from large ASTs. This study proposes a novel Subtree-based Attention Neural Network (SANN) to address these gaps by integrating different components: an optimized sequential subtree extraction process using Genetic algorithm optimization, a two-way embedding approach, and an attention network. We investigate the effectiveness of SANN by applying it to two different tasks: program correctness prediction and algorithm detection on two educational datasets containing both small and large-scale code snippets written in Java and C, respectively. The experimental results show SANN's competitive performance against baseline models from the literature, including code2vec, ASTNN, TBCNN, CodeBERT, GPT-2, and MVG, regarding accurate predictive power. Finally, a case study is presented to show the interpretability of our model prediction and its application for an important human-centered computing application, student modeling. Our results indicate the effectiveness of the SANN model in capturing important syntactic and semantic information from students' code, allowing the construction of accurate student models, which serve as the foundation for generating adaptive instructional support such as individualized hints and feedback.

CCS CONCEPTS

• **Applied computing** → **Education**.

KEYWORDS

program analysis; code representation; static analysis; algorithm detection; program correctness prediction

ACM Reference Format:
Muntasir Hoq, Sushanth Reddy Chilla, Melika Ahmadi Ranjbar, Peter Brusilovsky, and Bita Akram. 2023. SANN: Programming Code Representation Using Attention Neural Network with Optimized Subtree Extraction. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23), October 21–25, 2023, Birmingham, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3583780.3615047

1 INTRODUCTION

Programming code representation is becoming an important area of research due to its growing application in various intelligent functionalities such as code classification [9, 43], bug detection [12, 39], and code summarization [1, 22]. A significant challenge posed by program analysis is the vast state space of programming code and the complex code structures that hinder the process of effectively capturing the semantic and syntactic information of programs [26, 28, 41]. Due to the success of different embedding and deep learning techniques in natural language processing (NLP) [19, 21, 37], these approaches are gaining popularity in representing source code. These approaches try to map from the vast state space of programs to condensed numerical vector forms. Various studies have shown the effectiveness of embedding program Abstract Syntax Trees (ASTs) in representing and analyzing programming code [5, 6, 31, 32]. However, multiple challenges are associated with this task, including effectively capturing the syntactic and semantic information of large programs with deep ASTs [26, 44]. The emergence of large language models pre-trained on massive amounts of data is a step toward addressing this issue [13]. However, they lack interpretability, which is critical in human-centered computational applications, including student modeling and personalization.

This study proposes a novel Subtree-based Attention Neural Network (SANN) with optimized subtree extraction using Genetic Algorithm (GA) for program representation and analysis. We employ GA optimization to dynamically split the program AST based on a particular modeling task. Using GA, we further optimize the maximum depth of each subtree, the maximum number of nodes in each subtree, and the maximum number of subtrees per program. The optimization process leads to capturing substantial structural and semantic information with fewer and smaller non-overlapping subtrees compared to extracting all the possible subtrees, which enables the model to analyze larger and deeper ASTs. These subtrees also help to capture semantic information more efficiently for their sequential extraction. A two-way embedding approach is then used to capture syntactic and semantic information across subtrees and programs. This is done by incorporating a subtree-based embedding for each subtree of a program and a node-based embedding for each subtree node. The optimized subtree extraction process, along with the two-way embedding approach, allows our model to capture task-specific structural information from code dynamically. Furthermore, an attention neural network combines subtree-level information according to its importance for the prediction task at hand. This attention mechanism allows the model to retain important subtree information in the final representation of the program. A combination of these approaches makes SANN effective in capturing useful information from program ASTs and analyzing larger ASTs while being interpretable. The embedded code can be further used in different prediction and analysis tasks such as code classification [28, 36], bug finding [24, 34], automatic hint generation [29, 30], etc.

The effectiveness of our model is validated by using it in two program analysis tasks using publicly available educational datasets in two different programming languages, Java and C. *Task 1* identifies the correctness of students' solutions for eight Java programming problems. It is based on the CodeWorkout [28] dataset, which consists of relatively small programs that represent correct and incorrect solutions for each problem, resulting in massive state space for possible programs. *Task 2* conducts algorithm detection in student solutions of C programming problems. It is based on the OJ dataset [26, 31], which contains correct solutions for 104 assignments that are relatively large and complex, resulting in larger and deeper ASTs. These two tasks provide insight into students' mastery of algorithms and their ability to solve a problem with specific requirements correctly. Taken together, these tasks show the effectiveness span of our model for analyzing relatively large and diverse ASTs.

The experimental results demonstrate that SANN competes successfully with several popular approaches reported and explored previously, including code2vec [6], ASTNN [44], TBCNN [31], GPT-2 [23], CodeBERT [13], and MVG [26] as well as some traditional Machine Learning (ML) techniques such as SVM, KNN, and XGBoost. We further demonstrate the interpretability of the SANN model in which a student's response to a coding problem is analyzed by investigating the most important subtree of the student's code AST affecting the results. Analyzing the syntactic and semantic information from student code structures provides insight into student programming skills [8]. Coupled with interpretability, it can further improve pedagogical approaches, provide adaptive course content recommendations, identify struggling students, and improve the overall process of learning programming [16]. Our model aims to generate accurate, effective, and trustworthy results by capturing important structural and semantic information from various sizes of ASTs while maintaining the ability to interpret its decision-making process.

In this study, we answer the following research questions:

- **RQ1:** *How well does SANN perform in predicting program correctness in sparse datasets with small ASTs?*
- **RQ2:** *How well does SANN perform in detecting algorithms from programs with larger and deeper ASTs?*
- **RQ3:** *How effective is the optimized subtree extraction process using GA?*
- **RQ4:** *Does the integration of subtree-based and node-based embedding (the two-way embedding approach) help in improving the performance of SANN?*
- **RQ5:** *Does the sequential subtree extraction process help in retaining semantic information?*

2 RELATED WORK

Different methods for representing programming code have evolved over time. Different data-based methods [33, 42] tried to map programs as functions where input data of the programs help to analyze the program with specific outputs. Sequence-based approaches [7, 46] tried to manipulate programs as natural language and applied different NLP techniques to analyze programs. Recently, different tree-based [5, 6, 31] and graph-based [26, 45] approaches have shown promising results in representing programs using ASTs and program graphs by adding edges to those ASTs.

At the same time, analysis of student programming data has gained popularity in recent years. Recently, AST-based techniques for analyzing programs have successfully captured program-specific information. For example, [35] used a contextual tree decomposition algorithm on program ASTs to generate automated hints. In [3], students' proficiency in various program-associated concepts was assessed using data-driven models. In another study [2], structural n-grams of varying sizes were used to represent the ordinal and hierarchical program structures. However, these approaches are not effective enough for analyzing large programs because their information extraction process is not equipped to capture complex code structures represented by large bodies of code.

In many recent studies, different embedding techniques have demonstrated their effectiveness in compressing programming code into a vector representation. [6] proposed a code2vec model in which each AST is split into paths between each pair of leaf nodes (referred to as context paths) while an attention mechanism is used to embed them into code vectors. It effectively predicted method names for programs written by professional software engineers. In [28, 39], the effectiveness of the code2vec model was shown using student programming data in program correctness prediction and bug detection tasks. In [32], an ast2vec model was introduced to generate vector embeddings from ASTs using a variational autoencoder by merging subtree information recursively. Other examples are code2seq [5] and ASTNN [44], where the information from ASTs is extracted using context paths and statement trees, respectively. [31] proposed a TBCNN model in which a convolutional kernel was used to capture AST information and embed them using max pooling. Recently, different pre-trained Transformer-based large models are also becoming popular in programming code analysis tasks [10, 13, 17, 23]. Another recent study [26] proposed a graph-based embedding model MVG that uses different graph information, i.e., control-flow, data-flow, and read-write graphs, and embeds them using a gated graph neural network with the help of max pooling. Both [26, 31] showed the effectiveness of their

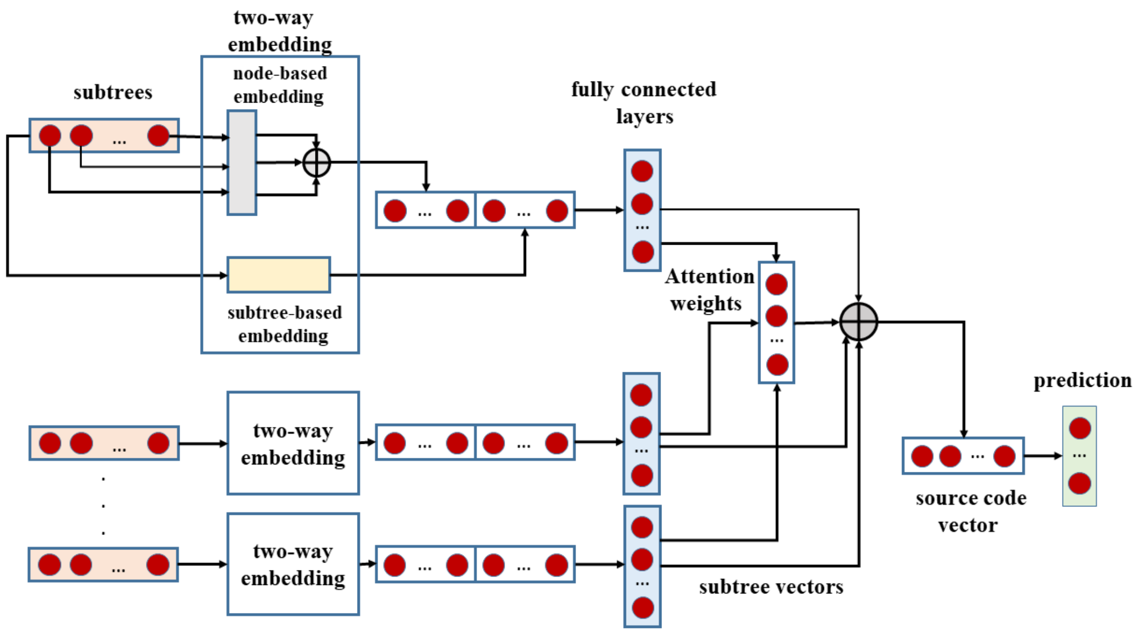


Figure 1: Architecture of code vector generation and prediction from extracted subtrees

models in algorithm detection tasks in student programs. Previous studies such as [26, 31, 32] have shown promising results with relatively small programs. However, they might be less effective with larger programs, which require preserving structural information for large and deep ASTs in the final vector representation. Effective representation can be hindered by merging structural information recursively or using max pooling, leading to the loss of syntactical and semantic information in the representation process [44]. Moreover, most of the previous studies [5, 6, 32, 44] use a static approach for AST splitting regardless of the tasks they are assigned. Splitting an AST into the same context paths, statement trees, or subtrees may limit the effectiveness of selecting the most task-relevant features.

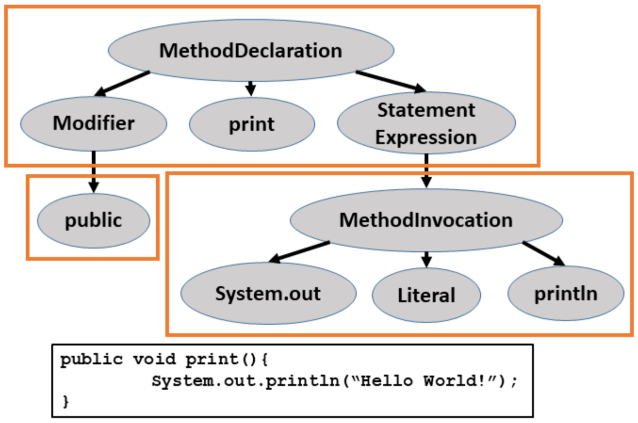
The proposed SANN model addresses gaps in the domain by dynamically capturing task-relevant information, aggregating node-level and subtree-structural information, and preserving long-term dependencies in large and deep ASTs. The optimized subtree extraction process, facilitated by genetic algorithms (GA), splits ASTs into non-overlapping subtrees of dynamically determined sizes based on the prediction task. This approach helps SANN generalize over larger and deeper ASTs while efficiently capturing semantic information through sequential subtree extraction and embedding. Unlike code2vec, which only embeds leaf nodes and context paths, SANN embeds individual nodes and subtrees, providing deeper insight into the local semantics of ASTs. The dynamic AST splitting process optimizes subtree size to retain task-specific information, while the attention mechanism ensures that important subtrees receive greater attention to prevent information loss. The architecture of the attention mechanism further enables us to identify and interpret the results and paves the way for creating adaptive instructional support based on the explanations. Overall, SANN's architecture effectively represents programming code by capturing essential syntactic and semantic information across various AST sizes in an interpretable manner.

3 METHODOLOGY

In this section, we introduce our Subtree-based Attention Neural Network (SANN) model. SANN uses an attention neural network to encode source code into vector forms. Subtrees are extracted from source code AST sequentially, where the size of the subtrees is optimized by a Genetic algorithm (GA). These subtrees are embedded using a two-way embedding approach to embed into corresponding subtree vectors. After that, an attention neural network generates the source code vector for each program. We can divide the whole model into two components: i) Optimized sequential subtree extraction using GA (Figure 3) and ii) Code vector generation and prediction (Figure 1). The following subsections delineate the design of the model.

3.1 Optimized Sequential Subtree Extraction Using Genetic Algorithm

An Abstract Syntax Tree (AST) represents the abstract syntactic structure of a source code [15]. Therefore, it can be used to understand the lexical semantics and syntactic structure of a source code (Figure 2). We extract subtrees from ASTs sequentially and optimize the process using GA as illustrated in Figure 3. ASTs are generated from source code using existing parser tools. We split an AST into non-overlapping subtrees at a granular level. This approach of extracting non-overlapping subtrees helps avoid repeating and redundant information in the input, which results in fewer and smaller subtrees and maximizes the amount of information that



```
public void print(){
        System.out.println("Hello World!");
}
```

Figure 2: AST with subtrees of max level 2 for a Java code

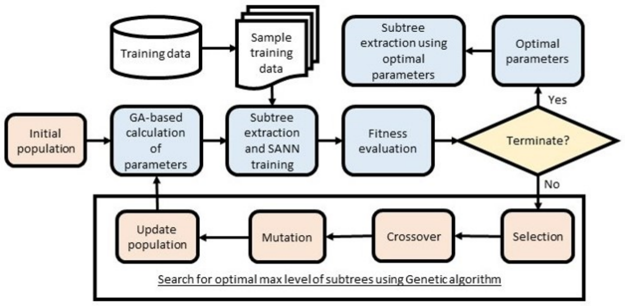

Training data
Sample training data
Subtree extraction using optimal parameters
Optimal parameters
Yes
Initial population
GA-based calculation of parameters
Subtree extraction and SANN training
Fitness evaluation
Terminate?
No
Update population
Mutation
Crossover
Selection
Search for optimal max level of subtrees using Genetic algorithm

Figure 3: Flowchart for subtree extraction process using GA

can be captured from an AST, given the computational resource constraints.

The maximum depth level for each subtree can be denoted as l. In Figure 2, the subtrees are shown for a simple Java code snippet where $l = 2$. In this case, each subtree will have a max level of two, and a node will not be considered a subtree's root if it is already present in another subtree. We choose the sequence of the root for each subtree sequentially using a breadth-first traversal. Each subtree is constructed using preorder traversal of nodes starting from the root. In Figure 2, we can see that the total number of subtrees is 3, and the length of the subtrees varies from 1-4. We need a fixed number of subtrees per program for the subtree-based embedding and a fixed length for all subtrees for the node-based embedding. Change of l influences the optimal max length of subtrees and the optimal max number of subtrees per program. We use GA to optimize l, max length of subtrees, and max number of subtrees for the dataset to overcome the huge computational time required to find the most optimal parameter by exploring the large parameter domain. In this study, we consider the last two parameters to have equal values to decrease training time and resources and refer to them as *max_subtree_size*.

GA is similar to the process of natural evolution, a meta-heuristic and stochastic optimization technique [18]. It is used in previous studies to optimize parameters where the parameter space is large [4, 38]. The main feature of GA is the population of "chromosomes". The process can be divided into six stages: initialization, fitness calculation, terminal condition check, crossover, selection, and mutation. At the initialization step, we set a chromosome arbitrarily as the initial value of l and *max_subtree_size* in the search space using binary bits representing genes of each value. The chromosome's fitness is set to the accuracy of the model's prediction. Crossover and mutation help to generate new solutions and introduce diversity in them. Fitness is determined for the newly generated chromosomes, and the model verifies the termination criteria. The highest accuracy is considered optimal fitness. Population size, number of generations, and gene length play an important role in finding the optimal solution. In this study, the aforementioned parameters are set to 4, 4, and 10, where the first 3 bits of the gene represent l, and the later bits represent *max_subtree_size*. Mutation probability and crossover probability are set to 0.6 and 0.2, respectively. The initial population is initialized randomly using the Bernoulli distribution. Likewise, ordered crossover, shuffle mutation, and roulette wheel selection are used.

In this approach, we can effectively split an AST into a sequence of subtrees, which can be used as the input for the later part of the model. Sequential extraction ensures the preservation of subtree-order and semantic information of the source code. Using GA, we can determine the maximum depth for each non-overlapping subtree extracted, the maximum number of subtrees, and the maximum length of each subtree for the given dataset to consider in the later phases. This helps to reduce the size and number of subtrees needed to gain insight into the local structure of ASTs.

3.2 Subtree Vector Generation

After the extraction of subtrees from an AST, the subtrees are embedded following a two-way embedding approach: subtree-based embedding to embed each subtree of the sequence and node-based embedding to embed each subtree node. This process requires the training of two embedding matrices: one for the subtree-based embedding and another for the node-based embedding, which can be denoted as *subtree_matrix*, and *node_matrix*, respectively.

$$subtree\_matrix \in \mathbb{R}^{|S|\times d};\ node\_matrix \in \mathbb{R}^{|X|\times d}$$

Here, S is the set of all subtrees in the subtree sequences, and X is the set of all AST nodes observed during the training time. The subtree and the node matrices are initialized with random values and are learned during the training process. SANN gets knowledge of similarities in subtrees and nodes from the subtree-based embedding and the node-based embedding, respectively. Thus, by incorporating these two embedding processes, the two-way embedding approach enables the model to identify similar code structures. The embedding size $d \in \mathbb{N}$ represents the dimension of vectors and is a hyperparameter of our model. The value of d is empirically determined depending on GPU memory, computing time, and model complexity. In literature, this value ranges from 100-500 [6]. The value of d can be different for subtree-based embedding and node-based embedding, but in this study, we kept the same value for both of the embeddings.

The two-way embedding of each subtree consists of an embedding of the subtree itself and an embedding of its nodes. These two embeddings are referred to as *subtree_embedded* and *node_embedded*, respectively. Let the subtree sequence for a source code be $S_i = \{s_1, s_2, ..., s_n\}$ and the node sequence of each subtree be $s_i = \{x_{i_1}, x_{i_2}, \ldots, x_{i_n}\}$ where $x_{i_j} \in X$ and $s_i \in S$.

We obtain the node vector for each subtree node by looking up its value in the *node_matrix*. After summing up the vectors representing each node in a subtree, we get *node_embedded*: the node-based embedding for a single subtree [5].

$$node\_embedded(s_i) = \sum_j node\_matrix(x_{i_j})$$

Similarly, we get *subtree_embedded*: the subtree-based embedding for each subtree by looking up its value in the *subtree_matrix*.

$$subtree\_embedded(s_i) = subtree\_matrix(s_i)$$

To merge the embedding vectors generated from the two-way embedding approach, we concatenate both the *node_embedded* and the *subtree_embedded* to a single embedded vector: $e_i \in \mathbb{R}^{2d}$.

$$e_i = [node\_embedded, subtree\_embedded] \in \mathbb{R}^{2d}$$

After that, a time-distributed fully connected layer generates the subtree vector sv_i from the embedded vector e_i by applying *tanh* activation function element-wise on the multiplication of e_i with the weight matrix W. This combines the node-level and subtree-level information.

$$sv_i = tanh(W \cdot e_i)$$

Here, *tanh* is the monotonic hyperbolic tangent activation function ranging from -1 to 1. This increases the model's expressiveness. W is the learned weight matrix for the fully connected layer where $W \in \mathbb{R}^{2d \times 2d}$. For convenience, the height of W is kept the same as the height of e_i, which is determined by the subtree vector length.

3.3 Source Code Vector Generation Using Attention

At this stage, SANN employs an attention neural network to condense all the subtree vectors for an AST into a single source code vector c. This attention mechanism determines a single scalar weight for each subtree vector sv_i and takes a weighted average to aggregate all the subtree vectors. An attention vector $av \in \mathbb{R}^{2d}$ is initialized randomly at the start and learned simultaneously while training. The attention weight a_i is calculated for each subtree vector sv_i by the normalized inner product of sv_i and the global attention vector av. Attention weight a_i for each subtree vector sv_i is calculated using a standard softmax function so that all the attention weights have a sum of 1, and the exponents in the softmax equation provide positive attention weights. Calculating an attention weight a_i using the softmax function is as follows:

$$a_i = \frac{exp(sv_i^T \cdot av)}{\sum_{j=1}^{n} exp(sv_j^T \cdot av)}$$

The weighted average of the subtree vectors using attention weights is then used to determine the vector representation of an entire source code snippet, $c \in \mathbb{R}^{2d}$. As determined by the attention network, this is a linear combination of the subtree vectors $\{sv_1, sv_2, \ldots, sv_n\}$ weighted by the importance of each subtree.

$$c = \sum_{i=1}^{n} a_i \cdot sv_i$$

Thus, while generating the source code vector, the most significant subtree receives the most attention.

3.4 Prediction

After the attention layer and source code vector generation, SANN employs an output layer with a softmax activation to classify code into respective class labels from the generated code vector, $c = \{c_1, c_2, \ldots c_{2d}\}$. The output layer generates a classification vector, $z = \{z_1, z_2, \ldots, z_{|Y|}\}$ for each code vector using a dense layer with output values for each class, where Y is the set of class labels. The final output after softmax activation, $o = \{o_1, o_2, \ldots, o_{|Y|}\}$ can be represented as the probability of the code snippet belonging to each class label y_i in a categorical manner. This is calculated as:

$$o_i = \frac{exp(z_i)}{\sum_{j=1}^{|Y|} exp(z_j)}$$

Therefore, o_i is the probability of a code snippet belonging to class y_i. The output of these probabilities by the softmax layer ranges from 0 to 1. Finally, SANN predicts the code snippet as the class with the highest probability of o_i.

4 EXPERIMENTS

We perform two classification tasks to investigate the effectiveness of our SANN model: i) Program correctness prediction (Task 1) and ii) Algorithm detection (Task 2). We also compare the performance of our SANN model with some state-of-the-art models from the literature as well as some traditional ML techniques.

4.1 Datasets

We use two educational datasets to conduct our experiments. For Task 1, we use the CodeWorkout dataset consisting of students' code submissions to Java programming problems on the CodeWorkout platform[1] labeled by their correctness: correct or incorrect [11]. We selected multiple student programs from 8 different problems from the Spring 2019 semester. In Task 2, we use the OJ dataset, which consists of student C programming solutions to different algorithmic problems collected from the Online Judge (OJ) platform[2] [31]. These programs are from 104 distinct algorithmic classes. These two datasets have been used in the literature for program correctness prediction [27, 28] and algorithm detection [26, 31].

Since uncompilable programs cannot generate ASTs, they are removed from the datasets. Table 1 shows some statistics for both datasets. We can see that the OJ dataset contains relatively larger and deeper ASTs than the CodeWorkout dataset. Also, compared to the CodeWorkout dataset, fewer data points are available per class (500 data points per 104 classes). The CodeWorkout dataset, on the other hand, contains shorter ASTs and a significantly larger number of data points per class (4000 data points per 2 classes). The CodWorkout dataset is particularly sparse since each class contains student-generated correct and incorrect solutions to eight different problems. Each dataset enables us to assess the affordances of our model for unique challenges presented by programming data. Using the CodeWorkout dataset, we can show the effectiveness of our model in capturing syntactic and semantic information to identify a

[1]https://codeworkout.cs.vt.edu/
[2]http://poj.org/

Properties	Task 1	Task 2
Dataset	CodeWorkout	OJ
Language	Java	C
Compilable problems	9403	52000
Classes	2	104
Max AST depth	22	76
Avg AST depth	8.2	13.4
Max AST nodes	464	7027
Avg AST nodes	76	190

Table 1: Dataset statistics

student's ability to solve a problem correctly in a sparse state space. Similarly, using the OJ dataset, we can verify the effectiveness of our model in identifying students' code structures, patterns, and mastery of algorithms from relatively larger programs.

4.2 Experimental Settings

We used pycparser[3] and javalang[4] tools to parse C and Java programming code into ASTs, respectively. For both tasks, we split our dataset into 80% training and 20% testing data in a stratified way to keep the same class ratios in both training and testing data. In the optimized subtree extraction process, we use 25% of the training data (20% of the whole dataset) to optimize the subtree size-related parameters using GA. For the CodeWorkout dataset, the max level for each subtree is set to 3, and both the maximum length of each subtree and the maximum number of subtrees per program are set to 100 using GA. Similarly, these values are set to 2 and 90 for the subtrees extracted from the OJ dataset. Due to time and resource constraints for training the deep learning models, we set the embedding size hyperparameter by changing its value manually to get the best output. The embedding size of both subtree-based and node-based embedding is set to 256 from {64, 128, 256}. As a result, each source code vector has a size of 512. We use a dropout [40] of 0.2 on the subtree vectors. During training the model, the Adamax optimizer [14] with the default learning rate of 0.001 is used to learn the weight of the matrices. The batch size is set to 128. The maximum number of epochs is set to 200 with an early stopping patience of 50 to stop overfitting the model.

4.3 Task 1: Prediction of Program Correctness

In this task, we use the CodeWorout dataset to predict the correctness of each program as correct or incorrect. Since the dataset has a class imbalance, we use precision, recall, and F1-score as evaluation metrics along with accuracy. We compare the performance of our model with traditional ML techniques such as SVM, KNN, and XGBoost. We use the TF-IDF feature extraction technique to prepare the numerical inputs for these traditional ML models [25]. We use 10-fold cross-validation for the hyperparameter tuning of SVM, KNN, and XGBoost. Apart from these models, we also compare SANN with the code2vec [6], ASTNN [44], GPT-2 [23], and CodeBERT [13] models from the literature.

[3]https://pypi.python.org/pypi/pycparser
[4]https://github.com/c2nes/javalang

- **Traditional ML techniques:** In addition to our SANN model, we also applied several other ML models to evaluate the performance of our model, such as SVM, KNN, and XGBoost. For SVM, the $kernel$ is set to rbf from $\{linear, poly, rbf\}$ and C to 10 from {0.1, 1, 10}. For KNN, $n\_neighbors$ is set to 10 from {5, 10, 20} and p to 2 (Manhattan distance). For XGBoost, gamma is set to 1 from {1, 5, 9} and max_depth to 6 from {3, 6, 10} [20].
- **code2vec:** code2vec splits an AST into all possible paths between different leaf nodes. It embeds the starting and ending nodes and the path between these two nodes in the embedding process. It employs an attention mechanism to merge path vectors into a unified code vector.
- **ASTNN:** AST-based Neural Network (ASTNN) is an RNN-based deep learning model representing programming code. It splits ASTs into small statement trees and encodes them using a bidirectional RNN.
- **GPT-2:** GPT-2 is a Transformer-based large language model with 1.5 billion parameters. In this study, we fine-tune GPT-2 for the given task.
- **CodeBERT:** CodeBERT is a Transformer-based model pre-trained on 6.4 million code (Python, Java, JavaScript, PHP, Ruby, Go). In this study, we fine-tune CodeBERT for the given task.

4.4 Task 2: Algorithm Detection

We use the OJ dataset in this task, where we detect the algorithm class for each class. Following previous studies [26, 31], we rely on accuracy as the evaluation metric for this task. To compare the performance of our model, we take into account some state-of-the-art models from the literature, including TBCNN [31], code2vec [6], ASTNN [44], GPT-2 [23], CodeBERT [13] and MVG [26].

- **TBCNN:** Tree-based Convolutional Neural Network (TBCNN) uses a convolutional kernel to capture structural information from program ASTs. It merges the information using max pooling to generate code vectors.
- **MVG:** Multi-View Graph (MVG) model is a graph-based deep learning model where control flow graphs, data flow graphs, read-write graphs, and a combined graph are used to extract information from programs with the help of a Graph Neural Network to represent programming code.

5 RESULTS & DISCUSSION

In this section, we demonstrate the experimental results concerning each research question. The first two research questions evaluate the performance of SANN in comparison to previous state-of-the-art models from the literature in addition to relevant traditional ML models. The last three research questions aim to investigate the effectiveness of different components of our model. Finally, a case study of how the results obtained from this model can be interpreted to inform adaptive educational technology is presented in the later part.

RQ1: *How well does SANN perform in predicting program correctness in sparse datasets with small ASTs?*

In Task 1, we use the CodeWorkout dataset, a small dataset with relatively short ASTs across eight problems, including correct and

Model	Accuracy	Precision	Recall	F-1 score
SVM[1]	0.74	0.71	0.70	0.70
KNN[1]	0.75	0.72	0.70	0.71
XGBoost[1]	0.77	0.75	0.74	0.74
code2vec[1]	0.79	0.76	0.76	0.76
ASTNN[1]	0.82	0.81	0.79	0.80
GPT-2[2]	0.77	0.76	0.74	0.75
CodeBERT[2]	0.88	0.85	0.87	0.86
SANN[1]	0.87	0.86	0.83	0.85

[1] Trained from scratch on CodeWorkout data
[2] Pretrained model finetuned on CodeWorkout data

Table 2: Performance comparison for program correctness prediction task on the CodeWorkout Dataset

incorrect solutions. Hence, we can see the performance of SANN compared to other models in shorter ASTs with vast state-space scenarios. From Table 2, we can see that XGBoost performs best among all traditional ML techniques. Programs like these contain different token names in different programs, i.e., variable names such as i, j, a, b, etc. This makes the prediction task difficult for traditional ML models as they depend on shallow semantic information based on token names. Both deep learning models outperform traditional ML models. SANN significantly outperforms all other models, including code2vec and ASTNN, with the highest accuracy, precision, recall, and F1-score with values of 0.87, 0.86, 0.83, and 0.85, respectively. SANN has slightly lower results than CodeBERT. However, a statistical significance test shows no significant differences between the results produced by SANN and CodeBert (p-value>0.05), although CodeBERT is pre-trained on 6.4 million code (1.5 million Java code). On the whole, our results indicate that the SANN model performs at the same level while being trained on a drastically lower amount of data (16000 data points). This strongly suggests that our model is capable of efficiently extracting important semantic information from the programming data.

RQ2: *How well does SANN perform in detecting algorithms from programs with larger and deeper ASTs?*

In Task 2, we evaluate the performance of SANN in the scenario of deeper and larger ASTs, where we use the OJ dataset to detect algorithms presented by programs. Retaining information in the vector representation process becomes more challenging for larger and deeper ASTs if we traverse and encode ASTs recursively in a bottom-up way or use max pooling to merge information. It can end up losing long-term context information. The optimized subtree extraction process and the two-way embedding approach to capture local syntactic and semantic information, coupled with the use of the attention mechanism to assign different weights to different subtrees based on their importance, SANN significantly (p-value<0.05) outperforms some of the prominent baseline models from the literature, including code2vec, TBCNN, GPT-2, CodeBERT (not pre-trained with C code) and MVG. with an accuracy of 96% (see Table 3). However, the ASTNN model significantly (p-value<0.05) outperforms SANN with an accuracy of 97%. It is worth mentioning that the classes in the OJ dataset have a dramatically lower number of data points per class (about 500) compared to the classes in the CodeWorkout dataset (4000 data points per class). Given the relatively large number of parameters in the SANN model, the low number of data points per class can result in overfitting of the model,

Model	code2vec[1]	ASTNN[1]	TBCNN[1]	GPT-2[2]	CodeBERT[2]	MVG[1]	SANN[1]
Acc (%)	90	97	94	82	95	94.96	96

[1] Trained from scratch on OJ data
[2] Pretrained model finetuned on OJ data

Table 3: Performance comparison for algorithm detection task on the OJ dataset

which affects its generalization error. We hypothesize that the performance of the SANN model can be improved by providing more data per class. We investigate this hypothesis by selecting 24 classes of algorithms and merging them into four superclasses based on their similarities. These superclasses consist of code based on string comparison, sorting, string replacement, and reversing order in a data structure. These classes have 3000 programs, which is higher than the usual 500 programs per class of the OJ dataset. We classify these superclasses using both SANN and ASTNN. SANN significantly outperforms ASTNN with an accuracy of 99% (p-value<0.05), whereas ASTNN shows an accuracy of 98%. In summary, the results show the effectiveness of SANN in efficiently capturing important structural and semantic information from relatively large and deep ASTs, enabling it to preserve long-term context information.

RQ3: *How effective is the optimized subtree extraction process using GA?*

In SANN, we use an optimized subtree extraction process using GA, where the size of non-overlapping subtrees extracted from each program is dynamically determined for each dataset and classification task. To know the effectiveness of this approach, we can consider another variant of SANN where the optimized subtree extraction process is replaced with a static subtree extraction process where all the possible subtrees of an AST are extracted. Extracting all possible subtrees from ASTs poses threats to the model's generalizability and the effectiveness of capturing syntactic and semantic information. Static subtree extraction is not adaptive to the assigned task and ends up with the same extracted subtrees every time. It also includes repetitive information as the subtrees overlap. It fails to generalize over larger and deeper ASTs, as the number of extracted subtrees increases with the number of nodes in the tree. Thus, It may lead to information loss by discarding a large number of subtrees during subtree-based embedding and also by discarding nodes during node-based embedding. These problems are solved by the optimized subtree extraction process using GA. To verify this claim, we train a variant of SANN with an all-possible subtree extraction process for both datasets. From Table 4, we can see that, for the CodeWorkout dataset, the accuracy values are not that different, as the programs in this data set are relatively smaller with shorter ASTs. Therefore, the performance does not vary significantly. For the OJ dataset, we can see that SANN with the all-possible subtree extraction process has an accuracy of 0.87, whereas the optimized SANN has the highest accuracy of 0.96. Since the programs in this dataset are relatively bigger with larger and deeper ASTs, we can see the effectiveness of the optimized subtree extraction process more clearly.

RQ4: *Does the integration of subtree-based and node-based embedding (the two-way embedding approach) help to improve the performance of SANN?*

We use a two-way embedding approach in SANN, integrating a subtree-based embedding to embed each subtree of a source code and a node-based embedding to embed each subtree node. We

Design alternatives	CodeWorkout	OJ
SANN+random subtree extraction	0.84	0.90
SANN+node-based embedding	0.75	0.89
SANN+subtree-based embedding	0.80	0.84
SANN+all-possible subtree	0.86	0.87
SANN	0.87	0.96

Table 4: Accuracy comparison of different SANN design alternatives

hypothesize that each embedding provides a unique source of information to our model. To verify this claim, we train our model with both of the datasets in three different ways: i) using a node-based embedding, ii) using a subtree-based embedding, and iii) using the two-way embedding approach. From Table 4, we can see that SANN with only node-based embedding surpasses SANN with only subtree-based embedding with an accuracy of 0.89 for the OJ dataset (Task 2), where the program algorithm is detected. This implies that the node-based embedding captures more information from the programs of the OJ dataset than the subtree-based embedding for this particular task. Since we are detecting algorithms from the code, node-based information such as name and type of variables, data structures, methods, etc., can convey more information to the model. However, SANN, with the two-way embedding approach, combines both node and subtree structural information and tries to capture more task-specific syntactic and semantic information effectively. Therefore, it has the highest accuracy among all three embedding design choices. We further verify this hypothesis on the CodeWorkout dataset (Task 1), where program correctness is predicted. For this dataset, SANN with the node-based embedding has an accuracy of 0.75, whereas the SANN with the subtree-based embedding has an accuracy of 0.80. This is justifiable since the subtree structures can provide more information about a correct program. For the CodeWorkout data, the two-way embedding approach outperforms the individual ones with an accuracy of 0.87. Therefore, even if node-based embedding is more important than subtree-based embedding or vice versa for a specific task, the two-way embedding approach is effective in different tasks by incorporating both.

RQ5: *Does the sequential subtree extraction process help in retaining semantic information?*

In the optimized subtree extraction process using GA, we sequentially extract the non-overlapping subtrees with a breadth-first traversal. We hypothesize that sequentiality helps us capture more semantic information about ASTs along with syntactic information. To test this hypothesis, we train our model following two approaches: with subtrees extracted i) randomly and ii) sequentially, as done in the original model. We compare the results using CodeWorkout (Task 1) and the OJ dataset (Task 2). From Table 4, we can see that the randomized subtree extraction process yields an accuracy of 0.84 for the CodeWorkout dataset and 0.90 for the OJ dataset, while, with the sequential subtree extraction process, SANN achieves the highest accuracy of 0.87 for the CodeWorkout dataset and 0.96 for the OJ dataset. Therefore, the sequential extraction of subtrees helps SANN to retain semantic information more efficiently.

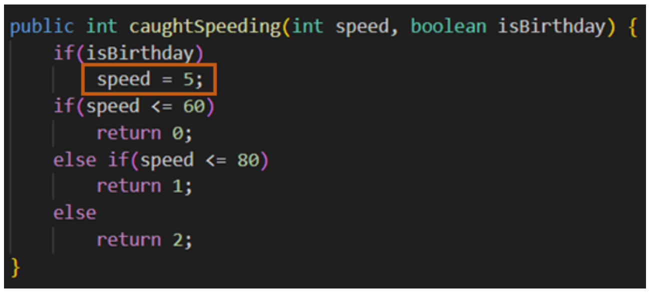

```
public int caughtSpeeding(int speed, boolean isBirthday) {
    if(isBirthday)
        speed = 5;
    if(speed <= 60)
        return 0;
    else if(speed <= 80)
        return 1;
    else
        return 2;
}
```

Figure 4: Most important subtree of an incorrect solution

Interpretability Case-Study *Can we meaningfully interpret the results obtained from SANN?*

The attention mechanism allows us to interpret the predictions done by SANN. We can understand which subtrees are the most important for a prediction by extracting their attention weights. We extract the subtree attention weights for an incorrect solution to a problem, named caughtspeeding, from the CodeWorkout dataset to verify this. Figure 4 shows an incorrect solution where *speed = 5* should be *speed -= 5* for the program to be correct. We extract the attention weights of the subtrees and see that the subtree with the incorrect statement has almost all the attention (92%) in predicting this program as incorrect. The interpretability of SANN offers numerous potential applications in the analysis of educational programming, including modeling students' programming competencies, bug detection, misconception detection, and adaptive feedback generation to support students. SANN can be a valuable tool in computer science (CS) education as it can help analyze student code and provide insights into students' understanding of the material. The interpretability of SANN paves the way for different educational programming analyses and applications, which cannot be afforded using models that are difficult to interpret. The SANN model can be used to provide more personalized and tailored instruction and adaptive feedback to students to support their learning.

The experimental results show that our SANN model effectively captures syntactic and semantic information from student programs using sequential subtree extraction, the two-way embedding approach, and the optimized subtree extraction process dynamically optimized through GA with regard to the specific prediction task at hand. Trained on relatively small amounts of data, our model can compete with large models pre-trained on excessively large datasets while maintaining interpretability. Coupled with its interpretable structure, our model can be used to build accurate models of students' programming competencies that provide us with insight into students' mastery of programming competencies.

6 CONCLUSION

This study explored a novel approach for programming code representation: a Subtree-based Attention Neural Network (SANN) with optimized subtree extraction using the Genetic Algorithm (GA). In the subtree extraction phase, SANN extracts subtrees of optimal size with regard to the specific prediction task from the program

AST sequentially, which get embedded following a two-way embedding approach. This helps SANN to capture more task-specific structural information dynamically. The two-way embedding approach incorporates subtree-based and node-based embedding to gain insight into the subtree structure-level and node-level information of ASTs. Additionally, an attention mechanism is used to generate the code vector for each program based on the importance of each subtree in the sequence. The effectiveness of SANN is investigated on two tasks: program correctness prediction and algorithm detection using two educational datasets in different programming languages, Java and C. Task 1 assesses how well our model extracts useful information and understands student ability to solve a problem correctly, as the dataset contains correct and incorrect solutions for different problems. Task 2 investigates the ability of our model to capture long-term dependencies and extract useful information from larger and deeper ASTs of student code. Our results demonstrate the effectiveness of our model in capturing important information from student programs. In addition, these results demonstrate our model's generalizing capability for large ASTs, identifying different algorithms from student code.

The experimental results suggest that SANN outperforms traditional ML models and other competitive baseline models from the literature, including code2vec, TBCNN, GPT-2, and MVG. SANN also outperforms ASTNN in Task 1 based on the CodeWorkout dataset containing Java programs but fails to outperform in Task 2 using the OJ dataset containing C programs. We hypothesize that this is due to the overfitting of the SANN model, given the higher number of parameters of SANN with regard to a low number of data points per class (about 500) in Task 2. We support our hypothesis by showing that SANN outperforms the ASTNN model when trained on a dataset where four superclasses are made by merging the six most similar classes from the OJ dataset into each superclass (3000 code per superclass). It is worth mentioning that SANN utilizes a two-way embedding approach, incorporating node-based and subtree-based embeddings, while ASTNN only employs node-based embedding. Results show that in Task 1, subtree-based embedding offers more valuable information than node-based embedding (Table 4). Consequently, SANN excels in tasks like Task 1, prioritizing structural AST information for code correctness prediction, giving it an edge over ASTNN. Similarly, in Task 2, where node information is crucial for algorithm detection (Table 4), SANN outperforms code2vec (using only path-based embedding similar to subtree-based) and even surpasses CodeBERT (pre-trained on 6.4 million code). In Task 1, there are no significant differences between SANN and CodeBert's performance despite CodeBERT's pre-training advantage.

We also performed a deeper study of our model to isolate the effectiveness of i) the optimized subtree extraction process using GA, ii) the two-way embedding approach, and iii) the sequential subtree extraction in capturing information from larger and deeper ASTs, capturing the similarities of local AST structures, capturing more semantic information. Our investigation indicates that various aspects of our model contribute to capturing important syntactic and semantic information from different sizes of ASTs in vast and sparse state spaces. Moreover, we explore a case study to show the interpretability of our model using an incorrect student program. Using the interpretability of the model enabled by the attention mechanism, the most influential subtrees can be extracted from the student code. This approach highlights specific parts of student programs where students may be struggling. Compared to other state-of-the-art models, SANN offers an easy-to-use interpretability feature without compromising the model's accuracy while being trained on relatively small datasets. These characteristics make SANN a great candidate for incorporation into CS education, as it can help analyze student code and provide insights into their understanding of the material. It can be used to provide more personalized and tailored instruction and adaptive feedback to students and to support their learning. Our main contributions are as follows:

- Proposing a novel SANN model with an optimized subtree extraction process for program representation that can represent deep and highly diverse ASTs.
- Validating our proposed model in two different program analysis tasks on learners' data to gain insight into their mastery of algorithms and ability to solve a problem correctly.
- Verifying the effectiveness of different aspects of the SANN model, including the GA algorithm for optimizing the subtree extraction process, the sequential extraction of subtrees, and the two-way embedding approach in capturing syntactic and semantic information from ASTs.
- Exploring the interpretability of the SANN model through a case study.

7 LIMITATIONS AND FUTURE WORK

One limitation of the proposed SANN incorporating GA is the increased training time. Specifically, the SANN+GA model requires approximately four times longer training time than the SANN that utilizes all possible subtrees. However, it should be noted that one of the primary applications of this model is to provide adaptive educational support for students. Once trained in an offline setting, the model can be utilized to predict various aspects of student learning in real time. As such, the increased training overhead of the SANN+GA will not impede its practical application. Furthermore, the trained model can be reused for new datasets in similar introductory programming classrooms, as the scope and scale of assignments tend to be consistent across different semesters or classrooms. In the future, we intend to evaluate a multi-task classifier that is built based on the SANN architecture and is trained on a combination of different datasets, optimizing a standard set of parameter values that can be used for every task and dataset. We hypothesize that this might lead to a reasonable trade-off between accuracy and training time for unseen tasks and datasets. Currently, our model uses fixed sizes for the embedding vectors. In the future, we intend to investigate the dynamic adaptation of vector sizes based on optimized subtree sizes to capture information from ASTs more efficiently. Finally, an interesting future direction is to build a pre-trained SANN model to ensure the highest accuracy and interpretability simultaneously. Through the attention mechanism, SANN paves the way to interpret student programs by understanding which part of a student program is more informative in analyzing a given task. We intend to explore the affordances of the attention mechanism to understand student programs and their learning and mistakes at a more granular level.

REFERENCES

[1] Ibrahim Abdelaziz, Julian Dolby, Jamie McCusker, and Kavitha Srinivas. 2022. Can machines read coding manuals yet?–A benchmark for building better language models for code understanding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 4415–4423.

[2] Bita Akram, Hamoon Azizolsoltani, Wookhee Min, Eric Wiebe, Anam Navied, Bradford Mott, Kristy Elizabeth Boyer, and James Lester. 2020. Automated assessment of computer science competencies from student programs with gaussian process regression. In *Proceedings of the 13th International Conference on EDM*. 555–560.

[3] Bita Akram, Hamoon Azizolsoltani, Wookhee Min, Eric N Wiebe, Anam Navied, Bradford W Mott, Kristy Elizabeth Boyer, and James C Lester. 2020. A data-driven approach to automatically assessing concept-level CS competencies based on student programs.. In *Proceedings of the Educational Data Mining in Computer Science Education (CSEDM) Workshop @ EDM*.

[4] Abdullah Al Mamun, Muntasir Hoq, Eklas Hossain, and Ramazan Bayindir. 2019. A hybrid deep learning model with evolutionary algorithm for short-term load forecasting. In *Proceedings of the 8th ICRERA*. 886–891.

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *Proceedings of the 7th ICLR*.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *ACM on Programming Languages* 3, POPL (2019), 1–29.

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems* 31 (2018).

[8] Robert Bodily, Judy Kay, Vincent Aleven, Ioana Jivet, Dan Davis, Franceska Xhakaj, and Katrien Verbert. 2018. Open learner models and learning analytics dashboards: a systematic review. In *Proceedings of the 8th International Conference on LAK*. 41–50.

[9] Nghi DQ Bui, Lingxiao Jiang, and Yijun Yu. 2018. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In *Proceedings of the Workshop at the 32nd AAAI Conference on Artificial Intelligence*.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] Stephen H Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: short programming exercises with built-in data collection. In *Proceedings of the ACM Conference on ITiCSE*. 188–193.

[12] Amir Elmishali, Roni Stern, and Meir Kalech. 2019. Debguer: A tool for bug prediction and diagnosis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 9446–9451.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-Trained model for programming and natural languages. In *Proceedings of the EMNLP*. 1536–1547.

[14] Anna Filighera, Tim Steuer, and Christoph Rensing. 2019. Automatic text difficulty estimation using embeddings and neural networks. In *Proceedings of the European Conference on Technology Enhanced Learning*. Springer, 335–348.

[15] Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. 2008. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software* 81, 3 (2008), 447–460.

[16] Josh Gardner and Christopher Brooks. 2018. Student success prediction in MOOCs. *User Modeling and User-Adapted Interaction* 28, 2 (2018), 127–203.

[17] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. Graphcodebert: Pre-training code representations with data flow. In *Proceedings of the 9th ICLR*.

[18] John H Holland. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT Press.

[19] Muntasir Hoq, Kazi Hasan Ibn Arif, and Mohammed Nazim Uddin. 2021. Local and Global Feature Based Hybrid Deep Learning Model for Bangla Parts of Speech Tagging. In *Proceedings of the 2nd INCET*. IEEE, 1–6.

[20] Muntasir Hoq, Peter Brusilovsky, and Bita Akram. 2023. Analysis of an explainable student performance prediction model in an introductory programming course. In *Proceedings of the 16th International Conference on EDM*.

[21] Muntasir Hoq, Promila Haque, and Mohammed Nazim Uddin. 2021. Sentiment analysis of Bangla language using deep learning approaches. In *Proceedings of the International Conference on Computing Science, Communication and Security*. Springer, 140–151.

[22] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on ASE*. IEEE, 135–146.

[23] Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. 2022. Fine-tuning GPT-2 to patch programs, is it worth it?. In *Proceedings of the Computational Science and Its Applications–ICCSA Workshops*. Springer, 79–91.

[24] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[25] Qing Liu, Jing Wang, Dehai Zhang, Yun Yang, and NaiYao Wang. 2018. Text features extraction based on TF-IDF associating semantic. In *Proceedings of the IEEE 4th International Conference on Computer and Communications (ICCC)*. IEEE, 2338–2343.

[26] Ting Long, Yutong Xie, Xianyu Chen, Weinan Zhang, Qinxiang Cao, and Yong Yu. 2022. Multi-View graph representation for programming language processing: an investigation into algorithm detection. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence*, Vol. 36. 5792–5799.

[27] Ye Mao, Farzaneh Khoshnevisan, Thomas Price, Tiffany Barnes, and Min Chi. 2022. Cross-Lingual adversarial domain adaptation for novice programming. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence*, Vol. 36. 7682–7690.

[28] Ye Mao, Yang Shi, Samiha Marwan, Thomas W Price, Tiffany Barnes, and Min Chi. 2021. Knowing "When" and "Where": temporal-ASTNN for student learning progression in novice programming tasks. In *Proceedings of the 14th International Conference on EDM*.

[29] Samiha Marwan, Bita Akram, Tiffany Barnes, and Thomas W Price. 2022. Adaptive immediate feedback for block-based programming: Design and evaluation. *IEEE Transactions on Learning Technologies* (2022), 406–420.

[30] Samiha Marwan, Joseph Jay Williams, and Thomas Price. 2019. An evaluation of the impact of automated programming hints on performance and learning. In *Proceedings of the ACM Conference on International Computing Education Research*. 61–70.

[31] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, Vol. 30.

[32] Benjamin Paassen, Jessica McBroom, Bryn Jeffries, Irena Koprinska, Kalina Yacef, et al. 2021. Mapping python programs to vectors using recursive neural encodings. *Journal of Educational Data Mining* 13, 3 (2021), 1–35.

[33] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *Proceedings of the International Conference on Machine Learning*. PMLR, 1093–1102.

[34] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[35] Thomas W Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating data-driven hints for open-ended programming.. In *Proceedings of the International Conference on EDM*. 191–198.

[36] Md Rahman, Yutaka Watanobe, Keita Nakamura, et al. 2020. Source code assessment and classification based on estimated error probability using attentive LSTM language model and its application in programming education. *Applied Sciences* 10, 8 (2020), 2973.

[37] Yafeng Ren, Yue Zhang, Meishan Zhang, and Donghong Ji. 2016. Improving twitter sentiment classification using topic-enriched multi-prototype word embeddings. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence.*

[38] Adarsh Sehgal, Hung La, Sushil Louis, and Hai Nguyen. 2019. Deep reinforcement learning using genetic algorithm for parameter optimization. In *Proceedings of the 3rd IEEE International Conference on Robotic Computing (IRC)*. IEEE, 596–601.

[39] Yang Shi, T Mao, Tiffany Barnes, Min Chi, and Thomas W Price. 2021. More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code.. In *Proceedings of the 14th International Conference on EDM.*

[40] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[41] Yu Su, Qingwen Liu, Qi Liu, Zhenya Huang, Yu Yin, Enhong Chen, Chris Ding, Si Wei, and Guoping Hu. 2018. Exercise-enhanced sequential modeling for student performance prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[42] Ke Wang. 2019. Learning scalable and precise representation of program semantics. *arXiv preprint arXiv:1905.05251* (2019).

[43] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on ASE*. IEEE, 87–98.

[44] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[45] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems* 32 (2019).

[46] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium.*