2023 IEEE 10th International Conference on Data Science and Advanced Analytics (DSAA) | 979-8-3503-4503-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/DSAA60987.2023.10302571

Towards Deep Learning Models for Automatic Computer Program Grading

Peter Nagy
Ontario Tech University, Oshawa, Canada
peter.nagy@ontariotechu.net

Heidar Davoudi
Ontario Tech University, Oshawa, Canada
heidar.davoudi@ontariotechu.ca

***Abstract*—Automatic grading of computer programs has a great impact on both computer science education and the software industry as it saves human evaluators a tremendous amount of time required for assessing programs. However, to date, this problem lacks extensive research from the machine learning/deep learning perspective. Currently, the traditional auto-grading systems are mostly based on test-case execution results. However, these approaches lack insight into the syntax and semantics of the codes, and therefore, are far from human-level evaluation. In this study, we leverage the power of language models pre-trained on programming languages. We introduce two simple deep architectures and show that they consistently outperform the shallow models built upon extensive feature engineering approaches by a high margin. We also develop an incremental transductive learning algorithm that only requires a single reference solution to a problem and takes advantage of the correct implementations in the set of programs to be evaluated. Furthermore, our human evaluation results show that the proposed approaches provide partial marks having a strong correlation with marks given by human graders. We prepare and share a dataset of C++ and Python programs for future research (Code and data are available at https://github.com/peter-nagy1/Deep-Grader).**

***Index Terms*—Automatic grading, Pre-trained language representation models, Siamese network, Transductive learning**

I. Introduction

The automatic and accurate grading of computer programs is of paramount importance due to its tremendous impact on educational programming courses. Compared to manually grading programming assignments, the use of such tools can immensely reduce the time of grading and help maintain impartial grading across all students. It is anticipated that the computer science field will have one of the greatest growth among different fields (26% between 2022–2027) in online enrollment over the coming years [1]. Therefore, adopting such tools can assist this demand, decrease the workload, and manage unprecedented class sizes.

The traditional auto-grading approaches require creation of an appropriate *test suite* for each question, facilitating their prompt evaluation. However, building such test suites demands ample time and effort. To ensure loophole-free test suites designers are required to perform rigorous analysis on the coverage of all possible combinations of inputs and outputs for each problem. This limits the complexity of questions that can be asked since a higher complexity question requires a larger test suite and more time to ensure its validity and completeness. Moreover, in the case that a program fails to compile/execute or for certain reasons fails to pass any test-cases, there is no way to evaluate the program using a test suite for potential partial marks (see Figure 1).

```cpp
#include <bits/stdc++.h>
using namespace std;

int main ()
{
    int n, minPrice, answer = 0;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        int x, y;
        cin >> x >> y;

        if (i == 0)
            minPrice = y;
        else
            minPrice = min(minPrice, y);

        answer += minPrice * x;
    }

    return 0;
}
```

Fig. 1: This program is only missing the "`cout`" statement, but it does not pass any test-cases. The human evaluators grade the program 3 and 4 out of 5. The proposed model predicts the grade of 3.32 out of 5.

Although effective auto-grading of computer programs is a demanding need, there are only few studies on the analytical side that make this mechanism more effective [2], [3]. To date, the majority of these works in machine learning are based on shallow models (e.g., LASSO, Support Vector Machine) designed on top of handcrafted features. While designing these features is a tedious task and may vary from one programming language to another one, they are far from deep insights, which are required for human-level assessment of programs.

Recently, advancements in Natural Language Processing (NLP) approaches based on pre-trained language models (e.g., BERT [4]) have shown significant performance improvement in many natural language understanding tasks [4]–[6]. Similar ideas are adapted in programming languages models such as CodeBERT [7], and UniXcoder [8]. However, to the best of our knowledge, these models have not been investigated for

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

automatic program grading. In this work, we leverage the advantages of models pre-trained on a large set of computer programs and adapt them for the problem of computer program grading.

The problem of grading computer programs is facing several challenges from the data mining/machine learning point of view. Fine-tuning existing models requires a large number of labeled samples (i.e., programs and their grades) which are not usually available. Furthermore, in order to provide maximal effectiveness, the proposed model should be *invariant* to questions, that is, the model trained for marking a particular question should be used for assessing other questions as well.

To that end, we propose different models and *end-to-end* solutions for the problem of automatic computer program grading. Our proposed frameworks take advantage of pre-trained models (i.e., CodeBERT [7], and UniXcoder [8]) built upon large collections of unlabeled programs. In order to make the models *invariant* to a specific question, we feed models with a question-specific correct solution (as a reference) and use appropriate loss functions (cross-entropy loss, contrastive loss [9]). We train the proposed models by utilizing *weakly* labeled samples, where the labels (i.e., correct/incorrect) are determined by the test suite results available for each program. That is, if the program fails on any test-case then it is incorrect, otherwise, it is correct. Our extensive experiments show that the proposed models trained on weakly labeled samples significantly outperform the shallow models. Moreover, our user study shows that the proposed approaches have a strong correlation with marks given by human evaluators. Finally, we propose an *incremental transductive algorithm* built upon a majority vote scheme, taking advantage of the correct implementations in the set of programs to be evaluated. This is similar to the scenario where an instructor is initially provided with a single solution to a problem, but upon discovering new correct solutions from the submissions, considers them as well as references to grade the rest.

The contributions of this paper can be summarized as follows:

- We propose end-to-end deep architectures for the problem of grading computer programs and show the possibility of training them using weakly labeled samples. To the best of our knowledge, this is the first work investigating the potential of pre-trained deep learning models for this problem.
- We design an incremental transductive learning approach taking advantage of correct programs in the test set based on a majority voting approach.
- We design comprehensive hand-crafted features capturing the program's syntax and structure, and compare the shallow models built using them with the proposed models.
- We conduct extensive experiments and show that while our proposed architectures need less supervision than shallow models built upon hand-crafted features, they also outperform the shallow models by a high margin.
- We conduct a user-study and find that partial grades predicted by the proposed models appear to have a strong correlation with human graders' marks.
- We collect and clean a dataset of C++ and Python programs, and release it for further research in evaluating the correctness of computer programs.

The rest of the paper is organized as follows. Section II goes over the related work. Section III presents the problem definition. Some background describing the pre-trained Transformer-based [10] models is presented in Section IV. Our proposed models are described in Section V. Section VI covers our feature-engineering baselines. Section VII presents our data, possible evaluation methods, the empirical results, and a user study to better evaluate the proposed models. Finally, Section VIII concludes the paper and suggests possible future work in the area.

II. Related Work

The grading of computer programs using machine learning has only been tackled with a feature-engineering approach in [2]. In this paper, they propose a grammar of features that is composed of six categories where each category specifies the type of code occurrences that are counted as features. The categories start from simple keyword counts to more complex data dependency and control-context between programming expressions. They then transform the features to make them invariant across questions. Finally, they experiment with multiple models including linear regression with L_1 regularization (LASSO), ridge regression, support vector machines (SVM), and random forest to train a supervised model for this task. The main drawback of this approach comes from the features that need to be redefined specifically to the programming language at hand. Algorithms for certain graph representations of code would need to be rewritten for every programming language as they mostly depend on specific keywords and structure of code which vary across languages. Furthermore, this design inevitably requires multiple correct solutions as reference to accurately grade responses which may not be feasible in deployment. In a deployment environment, this design necessitates a test suite to acquire a number of correct programs in order to accurately grade responses to an unseen question.

Outside of machine learning, few research papers proposed grading computer programs based on the number of test-cases passed [11], [12]. However, this approach does not evaluate the program's efficiency and style of writing. Furthermore, a minor error in an almost correct program can make it fail all test-cases judging it as completely incorrect by this approach (see Figure 1). On another note, a recent paper [13] has suggested grading computer programs by comparing their execution paths to a reference solution within a problem. In the case, that the proposed system finds a semantically different execution path it deems the program incorrect. On the downside, this approach cannot provide a partial grade to a program.

Other related papers focus on finding errors and providing feedback [14], [15] or correcting them [16]–[18] in computer programs. The task of plagiarism detection [19] or code

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

clone detection [20], [21], involving the detection of similar programs, can also be linked to our task, as likewise, we attempt to deduce the similarity between program pairs. Some have approached their problem with recurrent neural network (RNN) models [16], [18]. While others used methods involving abstract syntax tree (AST) representations of programs and developed complex algorithms such as AutoGrader [15] and CLARA [17] to solve their problem.

Recently, great innovations have emerged in the natural language (NL) processing field due to the Transformer [10] architecture, more specifically, by the use of pre-trained large language models such as BERT [4], BART [6], T5 [5], etc. Similar models have surfaced in the programming language (PL) field based on their NL counter-part including CodeBERT [7], GraphCodeBERT [22], UniXcoder [8], PLBART [23], CodeT5 [24], etc. Some of these models have incorporated multiple input modalities to pre-train their model other than the PL source code such as NL in the form of code comments and graph representations of code in the form of AST [8], [24] or data flow [22].

To measure the capabilities of a wide spectrum of state-of-the-art PL models, we decided to leverage CodeBERT and UniXcoder into our model architectures, since these pre-trained models have been proven to perform outstandingly on many code-related downstream tasks considering the current best approaches [8]. To the best of our knowledge, CodeBERT is the earliest implementation of a pre-trained Transformer-based model dedicated for Programming Language (PL) processing. It is then continuously used by future research in the field as a strong baseline [8], [22], [25]–[27]. Following CodeBERT, researchers introduce graph representations into their model, and amongst them, UniXcoder shows to outperform the previously mentioned rivals on most downstream tasks [8].

III. Problem Definition

The true assessment of a computer program's grade requires a thorough analysis by external expert evaluators. However, building a large dataset of programs paired with their corresponding true labels (i.e., grades) is an extremely time-consuming task. To that end, we utilize *weakly-labeled* samples where the label is determined by a test suite created for each given problem. A program is deemed *correct* or positive if it passes all test-cases within the test suite, otherwise, it is deemed *incorrect*. More formally, the *j'th* solution for problem *i* is denoted by $P_j^i =< s_{j,1}^i, s_{j,2}^i, \ldots, s_{j,m_j}^i >$, where $s_{j,k}^i$ is the *k'th* token in the *j'th* solution for problem *i*. The label $l_j^i = 1$ if P_j^i is correct, and $l_j^i = 0$ if P_j^i is incorrect. Our training dataset is composed of programs and corresponding labels as follows:

$$D = \{(P_j^i, l_j^i) | i = 1, 2, \ldots \; j = 1, 2, \ldots\} \tag{1}$$

Furthermore, in the following sections, we assume P_+^i is a correct solution for question i, and P^i is a solution that could be correct or incorrect.

Problem Statement: Given a training set D, the goal is to build a model that takes a correct program/solution $\widehat{P}_+^q \notin D$ for question q, and estimates the correctness of an unseen solution $\widehat{P}^q \notin D$ for an unseen question q by assigning a partial mark to $\widehat{P}^q$ (between 0 and 1). It is worth mentioning that partial marks can be used for binary decisions (i.e., if the partial mark is greater than or equal to 0.5 it is correct, and otherwise, it is incorrect).

IV. Background

A. CodeBERT

CodeBERT [7] is a pre-trained programming language model utilizing an *encoder-only* architecture identical to that of the base RoBERTa [28] model. RoBERTa is derived from BERT [4], both using a Transformer-based architecture. CodeBERT is trained using the Masked-Language Modeling (MLM) objective, consisting of predicting randomly masked code tokens, as well as the Replaced Token Detection (RTD) objective, aiming to predict whether a token at a specific location appears in the original program or it has been replaced. Both of these objectives were originally developed for natural language models and then adapted to programming language models. The adaptations mainly involve addressing both *bi-modal* (i.e., training based on pairs of natural language and programming language examples) and *unimodal* (i.e., training upon pairs of two programming language instances) data. The model has a total of 125M parameters which are trained on the CodeSearchNet [29] dataset with six programming languages including Go, Java, Javascript, PHP, Python, and Ruby.

B. UniXcoder

UniXcoder [8] is a general pre-trained model that proposes an *encoder-only*, a *decoder-only*, and an *encoder-decoder* framework for a range of code-related tasks. The different behavior modes (encoder-only, decoder-only, and encoder-decoder) are enabled using an input prefix (`[Enc]`, `[Dec]`, and `[E2D]`). The model takes advantage of code comments and Abstract Syntax Tree (AST) to enrich code representations. While code comments could be helpful for translation-related tasks (e.g., translating from one programming language to another one), abstract syntax trees retain all structural information of codes in embeddings. They utilize multiple pre-training tasks including Masked Language Modeling (MLM) in the encoder-only mode, Unidirectional Language Modeling (ULM) for the decoder-only mode, where the task consists of predicting the next token conditioned on previous tokens. They further employ a DeNoiSing (DNS) objective in an encoder-decoder mode which consists of predicting random masked spans. The model also learns semantic embeddings from multi-modal code using Multi-modal Contrastive Learning (MCL), and Cross-Modal Generation (CMG). The former employs a contrastive learning approach based on the SimCSE [30] framework and the latter consists of generating code comments. They also mention that while AST representations are crucial in pre-training the model, they are not required when fine-tuning the model to specific downstream tasks. The model includes 125M trainable parameters and follows a Transformer-based model architecture. The UniXcoder model

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

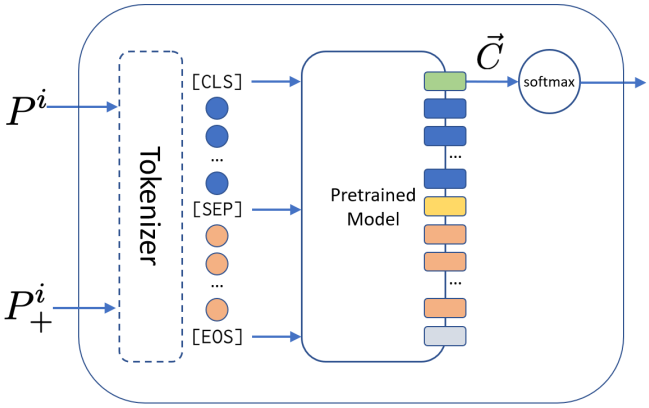


Fig. 2: Deep Grader Architecture

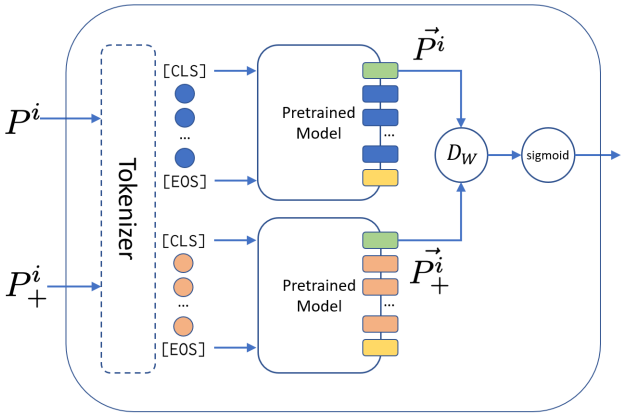


Fig. 3: Deep Siamese Grader Architecture

was trained on the CodeSearchNet [29] dataset and additionally, on the C4 dataset [5], consisting of English text, with a denoising objective to help the model with various bimodal tasks.

V. Proposed Methods

We propose two architectures leveraging these pre-trained models for programming language grading.

A. Deep Grader

Figure 2 shows the Deep Grader architecture. We create our dataset by combining all coding submissions with a random correct submission belonging to the same question. In this way, we can achieve question-independent grading since the model only needs to learn to differentiate the two input submissions irrespective of the problem. The programs are then processed through a pre-trained *tokenizer* which joins the two programs into one sequence padded and truncated to a fixed sequence length.

The token sequence starts with the `[CLS]` special token followed by two tokenized programs separated by a `[SEP]` token and ends with the `[EOS]` token. For example, a program pair (P^i, P^i_+) is encoded as:

$$< \texttt{[CLS]}, P^i, \texttt{[SEP]}, P^i_+, \texttt{[EOS]} > \tag{2}$$

Note that P^i_+ is a correct solution for question i, and P^i is a solution that could be correct or incorrect.

The output of the pre-trained models incorporates contextual vector representation for each token and an aggregate representation of the sequence is captured in the `[CLS]` token embedding denoted by $\vec{C}$. The pre-trained model's output (i.e., $\vec{C}$) is then fed into a softmax layer to provide the final output.

The model is trained using the cross-entropy loss:

$$L\left(W, Y, P^i, P^i_+\right) = -(Y^i \log(\widehat{Y}) + (1 - Y^i) \log(1 - \widehat{Y})) \tag{3}$$

where W is the model parameters, $\widehat{Y}$ is the predicted probability of P^i's correctness, and Y^i is the class label, which is 1 if P^i is a correct program, and 0 otherwise.

B. Deep Siamese Grader

Figure 3 shows the Siamese-based architecture for grading computer programs. The anchor program P^i and correct program P^i_+ are tokenized independently and fed into two separate instances of the pre-trained models sharing weights with each other. The model uses the contrastive loss function [9] to learn the parameters. That is, the outputs of the models, which represent low-dimensional embedding of the programs P^i, and P^i_+, denoted by $\vec{P^i}$, and $\vec{P^i_+}$ respectively, are compared using the parameterized Euclidean distance function:

$$D_W\left(\vec{P^i}, \vec{P^i_+}\right) = \left\|\vec{P^i} - \vec{P^i_+}\right\|_2 \tag{4}$$

The distance is then combined with the labels assigned to the pair of programs to generate the loss function:

$$L\left(W, Y, P^i, P^i_+\right) = (Y)\frac{1}{2}\left(D_W\right)^2 + (1 - Y)\frac{1}{2}\left\{\max\left(0, m - D_W\right)\right\}^2 \tag{5}$$

where Y is the label representing the anchor program P^i's correctness, 0 if incorrect and 1 if correct, W represents the model weights, D_W is the parameterized Euclidean distance function, and m is the margin or radius.

The stochastic gradient approach is used to update the weights. In inference phase, we normalize the distance between the anchor and correct programs using a sigmoid function to generate the model's predictions.

C. Majority Vote Grading

As we have a trained model, we can predict the grade of a new unseen program $\widehat{P}^i$ using the set of correct solutions $\mathcal{P}^i_+$ more effectively. In order to take advantage of all potential solutions in $\mathcal{P}^i_+$, we utilize a *majority voting* scheme. Algorithm 1 outlines the general procedure. The algorithm takes the trained model, the program to be evaluated for question i, denoted by $\widehat{P}^i$, and the set of correct programs for question i, denoted by $\mathcal{P}^i_+$, in order to predict the grade for $\widehat{P}^i$. While looping through each correct program $\widehat{P}^i_+ \in \mathcal{P}^i_+$, first, we pair the program $\widehat{P}^i$ with a correct program $\widehat{P}^i_+ \in \mathcal{P}^i_+$, then,

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Majority Vote Grading ($MajVote$)

Input: $\mathcal{M}$, trained model
Input: $\widehat{P}^i$, the program to be evaluated for question i
Input: $\mathcal{P}^i_+$, the set of correct programs for question i

1: $\mathcal{L} \leftarrow \{\}$
2: **for each** $\widehat{P}^i_+ \in \mathcal{P}^i_+$ **do**
3: $\quad seq \leftarrow Tokenizer(\widehat{P}^i, \widehat{P}^i_+)$
4: $\quad \widehat{y} \leftarrow \mathcal{M}(seq)$
5: $\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{\widehat{y}\}$
6: **end for**
7: $y_{pred} \leftarrow Mode(\mathcal{L})$ ▷ predicted label for $\widehat{P}^i$
8: $s_{pred} \leftarrow \underset{y \in \mathcal{L}}{Average}(prob(y))$

Algorithm 2 Incremental Transductive Grading

Input: $\mathcal{M}$, trained model
Input: $\mathcal{P}^i$, the set of programs to be evaluated for question i
Input: P^i_+, a correct program for each question $i \in \mathcal{Q}$

1: **for each** $i \in \mathcal{Q}$ **do**
2: $\quad \mathcal{P}^i_+ \leftarrow \{P^i_+\}$
3: $\quad$ **repeat**
4: $\quad\quad$ **for each** $p \in \mathcal{P}^i$ **do**
5: $\quad\quad\quad (y^i_{pred}, s^i_{pred}) \leftarrow MajVote(\mathcal{M}, p, \mathcal{P}^i_+)$
6: $\quad\quad\quad$ **if** $y_{pred} = 1$ **then** ▷ correct label
7: $\quad\quad\quad\quad \mathcal{P}^i_+ \leftarrow \mathcal{P}^i_+ \cup p$
8: $\quad\quad\quad$ **end if**
9: $\quad\quad$ **end for**
10: $\quad$ **until** $\mathcal{P}^i_+$ is unchanged
11: **end for**

tokenize the pair, and compute the label of $\widehat{P}^i$ using our trained model $\mathcal{M}$. Finally within the loop, we add the predicted label for $\widehat{P}^i$ to a set $\mathcal{L}$. Lastly, we compute the mode of labels in $\mathcal{L}$ to obtain the final weak label for $\widehat{P}^i$ (this weak label will be used in the next section where we perform incremental grading). The $\widehat{P}^i$'s predicted grade is the average of grades predicted based on all $\widehat{P}^i_+ \in \mathcal{P}^i_+$.

D. Incremental Transductive Grading

In section V-C, we assumed that there is a set of correct solutions for each question i. However, in a more realistic setting, we have only one correct solution in $\mathcal{P}^i_+$. In order to address this issue, we design an incremental transduction grading scheme, taking advantage of other potential correct solutions in the set of programs to be evaluated. The procedure is demonstrated by Algorithm 2. While this approach assumes only one correct program initially in the correct set $\mathcal{P}^i_+$ (line 3), it expands this set with newly found correct solutions using the majority vote algorithm's predictions. In particular, this algorithm requires a trained model, the set of programs to be evaluated for each question $i \in \mathcal{Q}$ (set of questions), and a single correct program P^i_+. For each question, we repeat the following process until the correct set for that specific question remains unchanged: First, for each program, we predict the program label based on the majority vote grading approach outlined in Algorithm 1 (line 5). If the program is predicted as a correct one (based on the predicted weak label), we add the program to our correct set $\mathcal{P}^i_+$ (line 7). We repeat the process until no new correct programs are found for the question.

VI. Feature Engineering Approaches

In order to show the effectiveness of the proposed deep learning models, we develop some baselines based on extensive feature engineering approaches. Our baseline models are inspired by the model proposed in [2]. The pipeline includes *feature extraction*, *feature invariant transformation*, and a *shallow prediction model*. Similar to our deep learning approaches, we train the models using a set of correct and incorrect solutions, but unlike our approaches, we utilize more than one correct solution for each question to evaluate a program's correctness.

Figure 4 presents the pipeline of our baseline model for a specific question. This model requires a large amount of correct responses to perform well since it utilizes all correct responses to compare and judge the correctness of a program. Hence, it aims to predict the correctness of a response to a new question given a large enough correct response set. This is quite an inconvenience since developers of new programming questions do not have a large array of correct responses to the question. They usually write one solution and grade the responses based on that. Producing multiple solutions is very expensive in terms of time and labour. Test-cases allow us to weakly label the responses in order to create a set of positive (correct) responses to a question. We can then extract lexical count-based features and semantic graph-based features from the programs. For each response, we compute distances between the given response (x) and each positive response (s^+) for a specific question, and aggregate the computed distances using the mean of the 10% minimum distances. Finally, we input these vectors to the classifier and the model outputs the label coinciding with the given response's correctness.

A. Feature Extraction

This section presents a set of features that can be extracted from the programming solutions. It is important to extract both *lexical* and *semantic* features from the programs. Thus, we focus on features capturing both aspects of the programs using *count-based* and *graph-based* features. The features emphasize key tokens, the overall structure, and data dependency between both statements and expressions, which are the essential properties of a computer program.

Count-based features: We first define different lexical count-based features separated into categories. Examples of keywords that are considered can be seen in Table I. The keywords adhere to the grammar of the specific programming language being used. We have found that similar features exist in both C++, Python, and other programming languages.

Graph-based features: In some cases, incorrect programs can have similar keyword counts to correct programs, however, they could be structurally different. In order to identify the

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Count-based feature categories

Categories	Examples
Basic	alphabet characters, numeric characters, ...
Loops	'for', 'while', ...
Data types	'int', 'float', ...
Operator	addition, subtraction, ...
Punctuation	semicolons, commas, ...
Bracket	curly, round, square, ...
Objects	'new', 'class', ...
Access modifier	'public', 'private', ...
Exception	'try', 'catch', ...
Conditional	'if', 'else', ...
Function	function declarations, function calls, ...
Input and output	'cin', 'cout', ...
Advanced	maximum integer, maximum array dimension, ...
Other	'using', 'include', 'define', ...

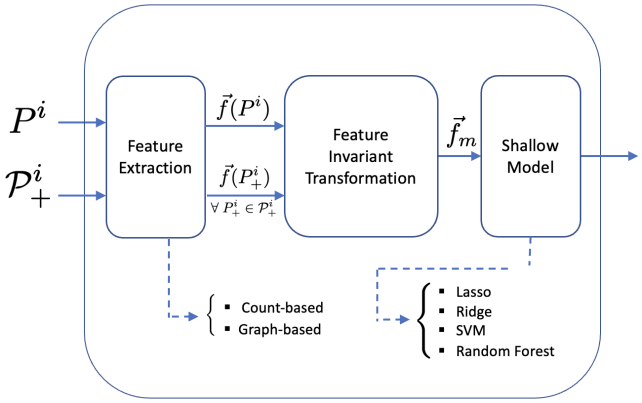


Fig. 4: Feature-based Model Pipeline

structure of the program and compare the programs based on their structures, we need graph-based features. As such, we define semantic features extracted from graphs representing the programs' structure. We define a *control-flow* graph, *statement data dependency* graph, and *expression data dependency* graph as well as the Abstract Syntax Tree (AST) graph (Figure 5).

The control-flow graph presents the overall structure of the program showing which statements are inside others. The statement data dependency graph presents the data flow between statements containing variables. The expression data dependency graph presents the data flow between variable expressions. The abstract syntax tree is a complex graph representation of source code widely used for parsing and program analysis [19]–[21], [31].

As we have these graphs, we extract different features from them such as minimums, maximums, and means of degrees, centrality, closeness, and clustering coefficients. We also consider the number of edges, the density, and the transitivity of the graphs as features.

B. Feature Invariant Transformation

In order to create a question-independent model, we followed the feature invariant transformation approach introduced in [2]. That is, we compute the distances between a solution (represented by a vector of feature) and the set of correct solutions for a specific question, then take the average of the distances and use that as the predictive model input for that solution. The intuition behind this is that two correct programs should be similar and return a low distance while the distance between an incorrect and a correct program should be higher. In the following definition, we assume that we have multiple correct solutions for each problem. However, in the case that we only have a single correct solution, we simply compute a single distance vector and use that as the input to our model as aggregation becomes unnecessary.

$$D^i = \{||\vec{f}(P^i) - \vec{f}(P^i_+)|| : \ \forall \ P^i_+ \in \mathcal{P}^i_+\} \tag{6}$$

$$\vec{f}_m = Mean(Min10(D^i)) \tag{7}$$

where $f(p)$ is the feature vector for program p and $Min10$ is the least 10% of the distances in D^+. In the approach from [2], they took the average of the top 25% of closest solution vectors. However, we found that the average of the top 10% of closest solution vectors will result in better performance. Thus, we set this hyperparameter to 10 when using multiple correct solutions per problem. $\vec{f}_m$ is used as the input of shallow models (see Figure 4).

C. Shallow Prediction Models

We tested many shallow models in our experiments and report the results of the best performing models as baselines. In particular, we utilize linear regression with both L_1 regularization (LASSO) and L_2 regularization (Ridge regression), Random Forest, and support vector machine (SVM) for training and evaluation purposes.

VII. Empirical Evaluation

A. Data Collection and Preprocessing

The data was retrieved from an open-access coding website called Codeforces that hosts competitive programming contests. A total of 200K Python programs and 200K C++ programs were retrieved from 1000 programming problems. The data was then cleaned to remove duplicate programs and problems with missing submissions, ending up with a total of 190K Python submissions from 985 questions and 168K C++ submissions from 992 questions. The website also provided labels (correct/incorrect) for each submission based on question specific test-cases created by the authors of the questions. A balance of correct and incorrect submissions was maintained during the retrieval process.

The collected dataset was then pre-processed. First, each program was ensured to be compilable using the `tree-sitter`[1] parsing tool. The code was then minimized and standardized using a custom modified version of the `pyminifier`[2] tool. The programs were stripped from blank lines, indentation was standardized, multi-line code was minimized to a single line, comments and doc-strings were

[1]https://github.com/tree-sitter/tree-sitter
[2]https://github.com/liftoff/pyminifier

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

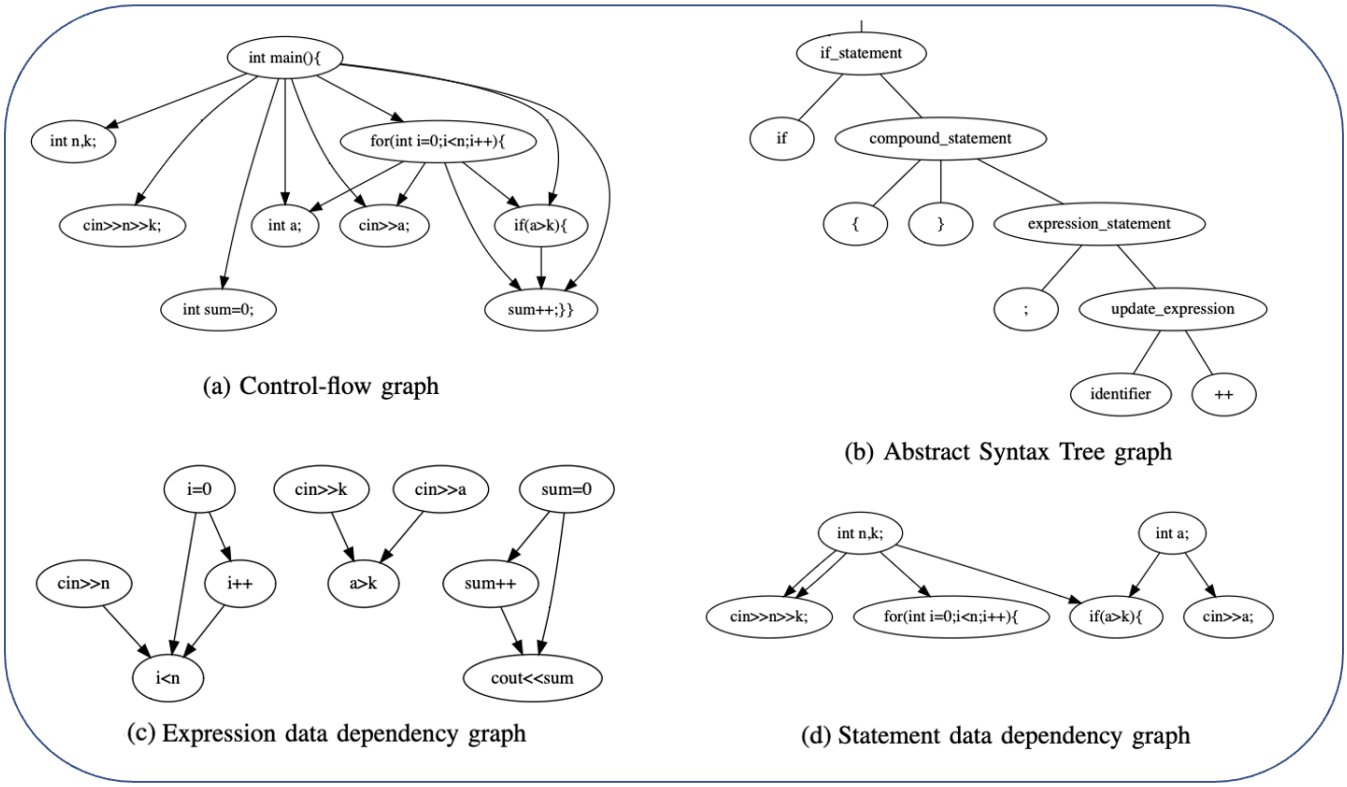


Fig. 5: Graph representations of programs

removed, and finally, unnecessary space characters around operators and other keywords were removed. The condensed programs range from 3 lines of code to 237 lines of code. Programs that still did not compile because of white-space inconsistencies or other formatting errors which could not be automatically fixed were manually adjusted.

The final dataset[3] includes problems ranging from a beginner's programming level requiring few simple control structures to an expert level including functions, recursion, multiple interlaced loops, many subtle conditional cases, and high dimensional array manipulations. The statistics of the data are presented in Table II.

TABLE II: Dataset Statistics

	Python	*C++*
No. of programs	190,451	168,110
No. of correct programs	96,934	82,430
No of incorrect programs	93,517	85,680
No. of questions	985	992
Avg. No. of programs/question	193.4	169.5
Avg. No. of correct prog./question	98.4	83.1
Avg. No. of incorrect prog./question	95.0	86.4

B. Baselines and Experimental Setup

We compare the proposed models with four shallow models (i.e., Random Forest, SVM, LASSO, and Ridge). The feature engineering pipeline is based on the previous state-of-the-art method [2] built upon the features discussed in Section VI. The set of programs is split along questions. In particular, 80% of questions will appear in the training set, 10% will appear in the validation set and the responses from the remaining 10% of questions will be used in our evaluation set. We optimize all hyper-parameters using the validation set. We run the experiments 3 times and report the performance of models, namely, Accuracy (Acc), F1-Measure (F1) in predicting weak labels, and the Correlation (Corr) between the model's predicted scores and the marks given by experts. We present the results for both Python and C++ datasets.

[3]Code and data are available at https://github.com/peter-nagy1/Deep-Grader

C. Comparing the Proposed Models with Feature Engineering Approaches

Table III shows the performance of the proposed models as well as the baselines where we use only one correct solution to evaluate an unknown solution. As can be seen, the proposed models outperform the baseline models consistently by a large margin for both Python and C++ datasets. We also see that in most cases Deep Grader outperforms Deep Siamese Grader especially when evaluated on Python data. Moreover, CodeBERT pre-trained model performs better for both Deep Grader and Deep Siamese Grader in term of F1 measure and Correlation (Corr). As another observation, although the pre-trained models (CodeBERT, UniXcoder) were never trained on C++ code, the results show similar performance for both

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

programming languages. This proves that the pre-trained models are highly flexible for this task and the pre-trained model trained based on one language can be fine-tuned and used in our framework for grading the solutions of a question in another language.

D. The Effect of Majority Vote Grading

Table IV illustrates the performance of the proposed models as well as the baselines where we use multiple correct solutions (in our Python dataset we have on average 98.4 correct solutions for each question) to evaluate unknown solutions for a question. Similar to Table III, the proposed models perform much better than the baselines consistently in terms of all performance measures for both Python and C++ datasets. Furthermore, in most cases, we observe that for this setting Deep Grader is superior to Deep Siamese Grader in terms of both datasets. It is worth mentioning that the baseline approaches based on the proposed framework in [2] feed the average of distances of all feature vectors of correct solutions to the feature vector of an unknown solution into the models and predict the unknown solution's label/score accordingly. While in the proposed majority vote approach, we predict the label for each pair of unknown and correct solution individually and then consider the majority of the predicted labels/average of the scores as the final label/score.

E. The Incremental Transductive Learning Analysis

Starting from a single solution, we can expand the correct set for each problem and utilize Majority Vote Grading to evaluate programs using multiple correct solutions. From Table V, we can discern that Incremental Transductive Grading helps improve the results in most cases. For example, Deep Grader (UniXcoder)'s accuracy (Acc) is of 73.9% and 76.6% when using a single correct solution and multiple correct solutions to evaluate an unknown solution for the Python dataset respectively. In this case, incremental learning achieves an accuracy of 75.7% which is an increase of 1.8pp when 2.7pp (76.6 - 73.9) is the room for improvement by using all correct solutions in the evaluation of an unknown solution.

F. Comparing with Human Grading

To compare the proposed models' grades with partial marks given by a domain expert, two experts have graded 200 random submissions from 10 random problems (with 10 incorrect and 10 correct solutions). The selected problems are of medium level difficulty necessitating few loops and conditionals to solve. The submissions are given a grade from 1 to 5 following the rubric described in Table VI. The average of the two graders' marks is considered as the final grade. The output of the proposed models that have been trained on weakly labelled data (i.e., the output of the *sigmoid* function) are normalized between 1 to 5, and the correlation between the partial marks (given by the domain experts) and the models' scores are reported in the performance tables. Table III and IV show significant improvement in the correlation measures when using the proposed methods. In particular, Deep Grader (CodeBERT) reveals a 16.2pp improvement over the best baseline (i.e., SVM) for the Python dataset, and Deep Siamese Grader (CodeBERT) is 19.9pp better than SVM for the C++ dataset. Table IV demonstrates a similar improvement when multiple correct solutions are available in inference time.

G. Interpretability Analysis

While the proposed models can provide partial marks for unknown programs, it is still not clear how partial marks are determined. In this section, we show how Local Interpretable Model-agnostic Explanations (LIME) [32] as a off-the-shelf explainablity tool can provide some insights into specific programming keywords that mostly influenced the decisions of our deep language models. Given an input to the model, LIME creates random perturbations to the input sequence and constructs a locally linear model using the ground-truth predictions obtained from the distinct perturbations. We applied LIME to explain the prediction of the model for both correct (Figure 6) and incorrect (Figure 7) programs and retrieved the tokens that contributed the most to the prediction of the model. In our experiments, LIME generated 100 perturbations of the program and yielded the 10 most influential features or tokens from the sequence.

In Figure 6, from the explanations given from LIME, we derive that the use of the `min` function was the most influential feature of the program followed by the `print` function, and the `split` function. The `min` function's importance can be justified by the fact that the output should be the minimum value from a list and it is the default approach, however, there are multiple ways of retrieving the minimum value from a list without using the `min` function. The `print` function is obviously important for the program to produce an output and there is not many alternatives. Finally, the `split` function is important to split the inputs over commas and it is the most logical approach. The rest of the features which are attributed lower weights do not influence the prediction of the model as much.

We also provide an example of a program from the same problem that is considered incorrect by the model (see Figure 7). Although, in this case, LIME does not provide as strong feature weights as previously to explain the prediction even though the prediction strongly indicates that this is an incorrect implementation. The most influential feature found which explains the sample program's incorrectness is the `if` token since the correct reference program does not contain an `if` statement.

H. Runtime Analysis

Figure 8 shows the average runtime of grading a program with the proposed models. The experiments were carried out on a cluster of 4 NVIDIA V100 GPUs. As can be seen, the grading time per program is fast (a fraction of a second).

VIII. Conclusion and Future Work

In this work, we presented question-independent models to grade computer programs. We designed two architectures and

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

TABLE III: Performance of models with single correct solution

Model	*Python* Dataset			C^{++} Dataset		
	Acc	**F1**	**Corr**	**Acc**	**F1**	**Corr**
Random Forest	62.3	63.7	51.0	64.4	61.8	55.7
SVM	63.9	68.7	61.0	65.2	66.0	55.8
LASSO (L1)	61.5	70.1	48.7	63.9	65.2	51.3
Ridge (L2)	61.3	68.9	54.3	65.2	66.0	51.0
Deep Grader (CodeBERT)	**74.5**	**74.4**	**77.2**	75.8	76.4	75.6
Deep Grader (UniXcoder)	73.9	71.9	77.1	**76.6**	76.8	75.2
Deep Siamese Grader (CodeBERT)	67.9	71.5	71.9	74.4	**77.1**	**75.7**
Deep Siamese Grader (UniXcoder)	67.8	71.7	73.8	73.6	76.1	75.4

TABLE IV: Performance of models with multiple correct solutions

Model	*Python* Dataset			C^{++} Dataset		
	Acc	**F1**	**Corr**	**Acc**	**F1**	**Corr**
Random Forest	69.9	71.2	72.9	70.7	71.1	57.4
SVM	65.6	71.5	71.0	69.5	70.9	62.0
LASSO (L1)	62.3	70.6	50.9	64.2	66.3	48.8
Ridge (L2)	64.0	70.8	55.6	66.7	67.9	56.4
Deep Grader (CodeBERT)	**77.0**	**77.3**	77.3	77.1	78.4	**76.8**
Deep Grader (UniXcoder)	76.6	75.2	**79.8**	78.2	**79.0**	76.2
Deep Siamese Grader (CodeBERT)	70.9	66.6	75.2	**80.0**	78.1	76.4
Deep Siamese Grader (UniXcoder)	70.3	65.0	74.7	77.4	75.9	75.3

TABLE V: Performance of models on Incremental Transductive Grading. In "()", the numerator shows improvement over the evaluation using a single correct solution and the denominator illustrates the difference in performance between the evaluation using multiple correct solutions and the evaluation using a single correct solution.

Model	*Python* Dataset			C^{++} Dataset		
	Acc	**F1**	**Corr**	**Acc**	**F1**	**Corr**
Deep Grader (CodeBERT)	75.0 (↑ 0.5/2.5)	**75.1** (↑ 0.7/2.9)	75.7 (↓ 1.5/0.1)	76.4 (↑ 0.6/1.3)	**77.6** (↑ 1.2/2.0)	**76.9** (↑ 1.3/1.2)
Deep Grader (UniXcoder)	**75.7** (↑ 1.8/2.7)	73.9 (↑ 2.0/3.3)	**78.1** (↑ 1.0/2.7)	76.7 (↑ 0.1/1.6)	77.0 (↑ 0.2/2.2)	76.2 (↑ 1.0/1.0)
Deep Siamese Grader (CodeBERT)	72.0 (↑ 4.1/3.0)	69.8 (↓ 1.7/-4.9)	75.9 (↑ 4.0/3.3)	**77.5** (↑ 3.1/5.6)	76.9 (↓ 0.2/1.0)	76.4 (↑ 0.7/0.7)
Deep Siamese Grader (UniXcoder)	71.9 (↑ 4.1/2.5)	69.5 (↓ 2.2/-6.7)	74.4 (↑ 0.6/0.9)	75.1 (↑ 1.5/3.8)	74.5 (↓ 1.6/-0.2)	75.2 (↓ 0.2/-0.1)

TABLE VI: Grading Rubric

Score	Interpretation
5	**Exceptional:** The code meets all of the requirements specified in the problem.
4	**Acceptable:** Necessary control structures and data dependencies are present with minor errors.
3	**Amateur:** Correct control structures are present with problems in data dependency.
2	**Unsatisfactory:** Demonstrating a slight understanding of the problem.
1	**Unrelated:** Solution is unrelated to the problem.

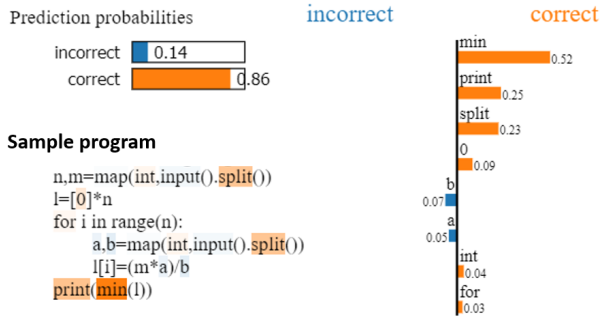


Fig. 6: LIME explanation of a correct implementation

fine-tuned the state-of-the-art pre-trained deep programming language models (i.e., CodeBERT [7], UniXCoder [8]) for the code grading task. Considering that the proposed approaches do not require any feature engineering, which depends on the programming language considered, they outperform the shallow models by a large margin. In particular, we developed a set of baselines based on both *count-based* and *graph-based* features and showed that they underperform compared to the proposed models. Furthermore, we developed a transductive learning algorithm taking advantage of correct solutions in the test set to make better predictions. Current deep learning models work as black boxes and do not provide any insights into the predicted grades. Therefore, we investigated the interpretability of these models with the use of Local Interpretable Model-agnostic Explanations. The feedback provided from LIME can easily be used by evaluators to understand the critical points of the program that influence its grade.

To the best of our knowledge, this is the first work towards

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.

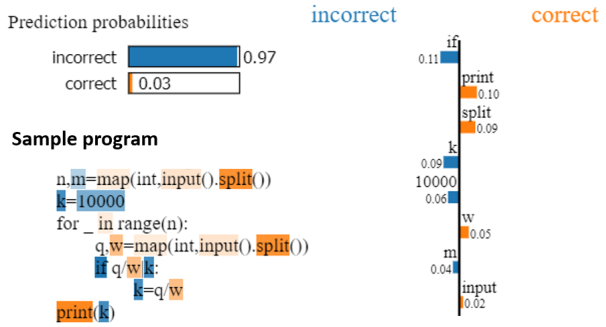


Fig. 7: LIME explanation of an incorrect implementation

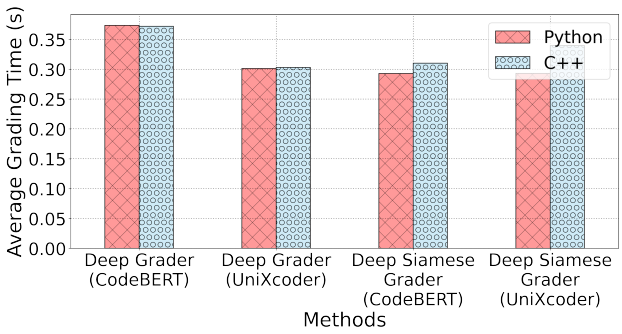


Fig. 8: Models' Grading Time per Submission

using deep learning models for the task of computer program grading. In the future, we plan on using Graph Neural Networks to leverage the code's structure (e.g., abstract syntax trees) directly in grade prediction. Additionally, developing semi-supervised models taking advantage of a large pool of unlabeled/weakly labeled programs and a smaller pool of programs evaluated by human graders in the training phase would be another interesting research direction. We hope that the proposed models and resources (including the datasets) which are prepared by this work pave the way for further research.

REFERENCES

[1] N. Foto, R. J. A. Rice, J. Stewart-Rozema, Q. Tomlin, and L. Zvereva, "Online education trends report," 2022.

[2] G. Singh, S. Srikant, and V. Aggarwal, "Question independent grading using machine learning: The case of computer program grading," in *KDD*, 2016, p. 263–272.

[3] V. Aggarwal, S. Srikant, and V. Shashidhar, "Principles for using machine learning in the assessment of open response items: Programming assessment as a case study," in *NIPS Workshop on Data Driven Education*, 2013.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019, pp. 4171–4186.

[5] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[6] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *ACL*, 2020, pp. 7871–7880.

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *EMNLP*, 2020, pp. 1536–1547.

[8] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *ACL*, 2022, pp. 7212–7225.

[9] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *CVPR*, vol. 2, 2006, pp. 1735–1742.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017, pp. 5998–6008.

[11] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *J. Educ. Resour. Comput.*, vol. 5, no. 3, p. 4–es, Sep. 2005.

[12] M. Wick, D. Stevenson, and P. Wagner, "Using testing and junit across the curriculum," in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 236–240.

[13] X. Liu, S. Wang, P. Wang, and D. Wu, "Automatic grading of programming assignments: An approach based on formal semantics," in *ICSE-SEET*, 2019, pp. 126–137.

[14] K. Rivers and K. R. Koedinger, "Automatic generation of programming feedback: A data-driven approach," in *The First Workshop on AI-supported Education for Computer Science (AIEDCS)*, vol. 50, 2013.

[15] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *PLDI*, 2013, p. 15–26.

[16] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *CoRR*, vol. abs/1603.06129, 2016.

[17] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *PLDI*, 2018, p. 465–480.

[18] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "Sk_p: A neural program corrector for moocs," in *SPLASH*, 2016, p. 39–40.

[19] M. Zheng, X. Pan, and D. Lillis, "Codex: Source code plagiarism detection based on abstract syntax tree." in *AICS*, 2018, pp. 362–373.

[20] H.-H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *IJCAI*, 2017, p. 3034–3040.

[21] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," *SANER*, pp. 261–271, 2020.

[22] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcode{bert}: Pre-training code representations with data flow," in *ICLR*, 2021.

[23] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *NAACL-HLT*, 2021, pp. 2655–2668.

[24] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *EMNLP*, 2021, pp. 8696–8708.

[25] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, "Treebert: A tree-based pre-trained model for programming language," in *UAI*, vol. 161, 2021, pp. 54–63.

[26] X. Wang, Y. Wang, Y. Wan, J. Wang, P. Zhou, L. Li, H. Wu, and J. Liu, "Code-mvp: Learning to represent source code from multiple views with contrastive pre-training," *ArXiv*, vol. abs/2205.02029, 2022.

[27] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," 2021.

[28] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019.

[29] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019.

[30] T. Gao, X. Yao, and D. Chen, "SimCSE: Simple contrastive learning of sentence embeddings," in *EMNLP*, 2021, pp. 6894–6910.

[31] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *ICSE*, 2019, pp. 783–794.

[32] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," in *KDD*, 2016, p. 1135–1144.

Authorized licensed use limited to: Ontario Tech University. Downloaded on November 10,2026 at 19:30:53 UTC from IEEE Xplore. Restrictions apply.