Automatic Grading of Student Code with Similarity Measurement

Dongxia Wang, En Zhang, and Xuesong Lu (✉)

East China Normal University, Shanghai, China
{dxwang,zhangen}@stu.ecnu.edu.cn, xslu@dase.ecnu.edu.cn

Abstract. Nowadays, online judges are extensively used for automatically grading student code. However, they grade code by only counting the number of passed test cases, which is not fair for assessing the overall quality of a code snippet. On the other hand, existing studies have used machine learning techniques for code grading. However, they usually require large amounts of labeled code to enable supervised learning and heavily rely on feature engineering. In this work, we design SimGrader, a code grading system that grades student code based on the measurement of similarity to the "good" code, and thus save the effort for code labeling. We extract three types of features to capture the overall quality of a code snippet, and design specific methods to enhance the feature discrimination, which facilitates the similarity measurement. We conduct extensive experiments to show the superiority of SimGrader over existing methods and justify the effect of the its system components. We deploy SimGrader to grade the student code submitted in an introductory programming course.

Keywords: Code grading · Discriminative feature · Contrastive learning · Tree edit distance

1 Introduction

Online judge (OJ) systems [23] have been extensively used in programming education [21,27,7,22,10]. The systems can automatically assess the correctness of student code by executing them with a set of pre-defined test cases, which greatly reduces teachers' burden of grading student code. However, for students, grading their code by simply counting the number of passed test cases is less informative and sometimes unfair with respect to the quality of the code. For instance, a code snippet passing all test cases may have awful code style or high time/space complexity, whereas a nearly correct code snippet may fail all test cases only due to one variable misuse. A good programming education should encourage students not only to write correct programs, but also to write concise programs with good style. As such, a fair code grading system should consider as many of the above factors as possible and give students a composite score to guide them in optimizing their code. A potential solution is to extract a set of code features pertaining to the grading factors and train machine learning models based on the features to predict the code grade.

Several related studies have investigated automatic code grading with machine learning techniques [19,18,20,14]. However, there exist following issues in these methods. First, they mainly construct the semantic features reflecting the functionality of a code snippet for grade prediction, e.g., expression features and data-dependency features [19,20], and rarely consider the features related to code style such as variable naming and indent use. These style features are actually crucial for assessing the quality of source code in terms of readability and ease of maintenance. Second, most existing methods manually construct the semantic features of source code, e.g., by counting the occurrences of key expressions and tokens [18,20], or calculating the number of specific nodes in the abstract syntax tree of a code snippet [14]. Manual feature engineering is ad-hoc for each programming question and may introduce noise into the features. Last, existing methods build supervised grading models and therefore require large amounts of labeled source code [19,18,20,16,14,11]. This not only imposes a lot of human labors but also leads to the underuse of massive unlabeled source code in the OJ system.

In this work, we implement the SimGrader system, which grades student code based on the measurement of similarity to the "good" code and therefore avoids the overhead of labeling the student code. For each programming question, "good" code is defined as a concise code snippet with good style to solve the question. Compared to labeling the student code, preparing the good code is much cheaper because the number of questions in an OJ system is often limited and we may find the good code from correct student submissions. For newly added questions, standard code snippets have to be composed for generating the test cases, which can be used as the good code. Note that each programming question may have multiple good code snippets. Then, we can generate the grade for a student code snippet based on its similarity (or distance) to the nearest good code. This is motivated by the human grading process where the teachers often compare a student code snippet with the nearest standard solution and give the grade based on the defects in the student code.

To implement SimGrader, at the core of the system is extracting the discriminative feature vector of student code that is used for code similarity measurement. We propose to extract the following three types of features. The first type are the static features such as the number of blank lines, improper spaces and indents, and the ratio of improperly named variables. This type of features mainly reflect whether a code snippet is concise and has good style. The second type of features are the runtime statistics such as execution time, memory usage and the percentage of passed test cases. This type of features mainly capture the efficiency and correctness of a code snippet. The last type are the semantic features of a code snippet, which reflect the functionality of a code snippet and thus are important for evaluating whether the code complies with the requirements of the programming question. Inspired by the recent advances in program representation learning [12,1,25,5], we propose to learn the features from massive student code using deep learning techniques rather than manually constructing them as in previous studies. To improve the discrimination of semantic features,

we further design a contrastive learning task followed by a fine-tuning task to obtain the final semantic feature vectors. The above three types of features are concatenated to form the complete feature vector. Finally, we calculate the similarity between the complete feature vector of a student code snippet and that of the nearest good code, based on which the code is graded. We deploy SimGrader in the OJ system used for an introductory programming course and demonstrate how we use it to grade student code.

Our contribution is summarized as follows:

- We design SimGrader, a system that grades student code based on the measurement of similarity to the "good" code and therefore avoids the expensive code labeling overhead. For similarity measurement, we propose to extract three types of code features including the static feature, runtime feature and semantic feature, which capture not only the functional information of a code snippet but also the conciseness and the style information. Moreover, the semantic feature is automatically learned from massive student code, avoiding the ad-hoc feature engineering and possible human-injected noise (See Section 3.1).
- To improve the discrimination of semantic features, we design a contrastive learning task to train the semantic feature vectors, so that the vectors of more similar code snippets are closer to each other. We further fine-tune the semantic feature vectors by predicting the closeness of each pair of code snippets, where the closeness is calculated based on the tree edit distance between the abstract syntax trees (ASTs) of the snippets (See Section 3.2).
- We conduct extensive experiments to show the effectiveness of SimGrader. We compare it with existing grading systems using the similarity measurement strategy as well as by training supervised prediction models with a small labeled dataset. We also conduct ablation studies to show the effect of the contrastive learning and fine-tuning sub-steps. The experimental results not only show the superiority of the extracted features but also justify the component design of SimGrader (See Section 4). We deploy SimGrader in an OJ system used for an introductory programming course and demonstrate some use cases (See Section 5).

2 Related Work

Existing studies mainly design supervised learning models for automatic code grading and heavily rely on feature engineering. For instance, Srikant and Aggarwal [19] construct six types of code semantic features by counting corresponding patterns. The patterns are extracted from the token sequence, the ASTs, the control flow graphs (CFG) and data dependency graphs (DDG) of a code snippet, and depend on each programming question. Based on the features, they train three regression models to predict code grade, including ridge, SVM and random forests. Later, Singh et al. [18] extend this work and propose a question independent method for code grading. They design a transformation which would transform a question specific feature matrix pertaining to a set of code

snippets into a structure invariant feature matrix. This matrix is further used as input to learn a question independent grading model. They use the similar feature engineering method with [19] and still need to label large amounts of code for learning the transformation. A related study [20] attempts to grade uncompilable code. Their focus is how to use feature engineering to extract semantic features from uncompilable code snippets.

Recent studies develop deep learning solutions for code grading. For instance, Orr and Russell [14] use feed-forward neural networks to predict code grade. However, they still use manually constructed features and need to label the code for supervised learning. Qin et al. [16] adopt a Bi-GRU network to learn from the intermediate code representation obtained using LLVM. They design a selection function to pick important features from the intermediate representation. The model is trained with labeled code snippets. A related but different study is conducted by Johnson-Yu et al. [9], where they design a model to find the unmarked code submissions that are mostly similar to the submissions that have been marked by a grader. Then they assign the unmarked submissions to the graders based on the similarity distribution, so that the efficiency of manual grading can be improved.

3 The SimGrader System

Figure 1 shows the overview of the SimGrader system consisting of three main steps. In the first step as shown in Figure 1(a), three types of features are extracted for each code snippet, including static features, runtime features and semantic features. Section 3.1 describes the details of feature extraction. In the second step as shown in Figure 1(b), two sub-steps including contrastive learning and fune-tuning are designed to enhance the discrimination of the semantic features. Section 3.2 describes the details of these two sub-steps. In the final step as shown in Figure 1(c), the concatenation of the three types of features is used to output the grading score for a code snippet based on similarity measurement with the good code. Section 3.3 describes this strategy. In the experiments, we also invite two experts to grade a small set of student code and show the results of supervised learning using the dataset.

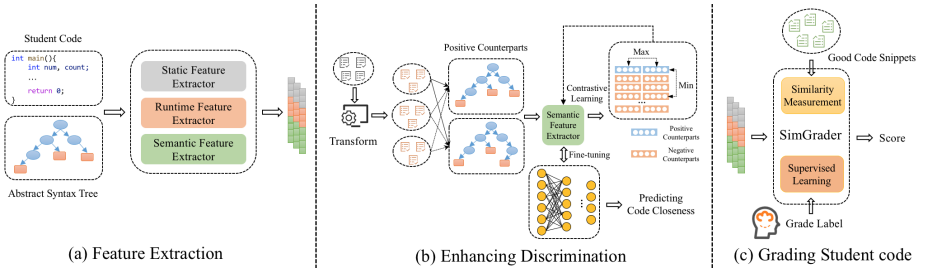
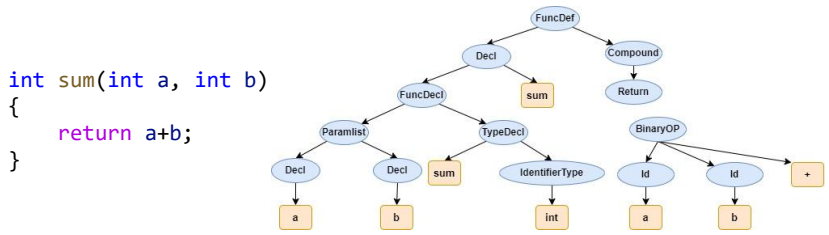


Fig. 1. Overview of the SimGrader system.

3.1 Feature Extraction

Static Features We construct the static features by traversing the text and the AST of a code snippet, without executing the code. The AST of a code snippet is an tree-based abstract representation of the grammatical structure, which is composed of semantic structure (internal) nodes and token (leaf) nodes. Figure 2(a) shows a simple code snippet and Figure 2(b) shows the corresponding AST. It can be observed that the leaf nodes correspond to the tokens in the code



(a) Code Statements (b) Abstract Syntax Tree

Fig. 2. A Code Snippet and its AST.

text, and the internal nodes indicate the semantic structure of (i.e., relationship between) the tokens. We use *pycparser*[1] to obtain the ASTs of student code. In addition, we also use static analysis tools such as *cpplint*[2] and *cppcheck*[3] to obtain some of the features. The static features obtained by traversing the code text and AST mainly capture the conciseness and style information of the code. The details of the features are described as follows:

- **Special lines:** Count the number of comments and blank lines. The intuition is that a code snippet with good style should have a certain number of comments and few blank lines.
- **Improper spaces and indents:** Count the number of improperly used spaces or indents (including too many or too few indents). We use *cpplint* to calculate the numbers.
- **Variables:** Count the number of variables and the number of times each variable is used. The intuition is that a concise code snippet should not have too many unnecessary and repeatedly used variables.
- **Variables naming:** Calculate the ratio of properly named variables over all variables, except the variables in the control statements. A properly named variable can be a word, an abbreviation, or the combination of words and abbreviations. Proper variable naming can enhance the readability and ease of maintenance of code [8,15].

[1] https://github.com/eliben/pycparser
[2] https://github.com/cpplint
[3] https://cppcheck.sourceforge.io/

- **Unused elements:** Count the number of unused variables and non-executed code lines. These elements commonly present in the code written by novice students. We use *Cppcheck* to calculate the numbers.
- **Cyclomatic complexity:** Count the number of judgement nodes in an AST. Common judgement nodes include *ForStatement*, *WhileStatement*, *IfStatement*, *BinaryOp* of boolean logic, etc. Cyclomatic complexity measures the complexity of code logic. For each programming question, a higher cyclomatic complexity indicates that the code is less concise and readable.
- **Halstead metrics:** Count the number of unique and total operators and operands, and use them in the Halstead formulas to calculate the metrics. These metrics mainly capture the static complexity of a code snippet.

Runtime Features The runtime features of the statistics obtained by executing student code with the test cases. The features capture the correctness and efficiency of the code pertaining to a specific programming question, which are described as follows:

- **Test cases:** Calculate the ratio of passed test cases over all test cases.
- **Execution time:** Calculate the maximum, average and minimum execution time of all test cases. The feature captures the time efficiency of code.
- **Memory usage:** Calculate the maximum, average and minimum memory usage of all test cases. The feature captures the space efficiency of code.

Semantic Features In previous studies [19,18,20], the semantic features pertaining to the functionality of a code snippet are obtained with feature engineering. This strategy needs to construct ad-hoc features for each question and may introduce noise into the features. A recent study [16] uses a Bi-GRU model [24] to learn from the token sequence of source code and output a distributed vector to represent the semantic features. However, learning from the token sequence leads to a significant effort to learn the syntactic nature of source code from scratch, which reduces the efficiency of learning the semantic features. Inspired by recent studies in program representation learning [12,1,25,5], we choose to learn the semantic features from the AST of a code snippet. The AST structure is shown to preserve the syntactic nature of a code snippet and therefore significantly lower the learning effort of semantic features [3,2]. In the current study, we experiment with two representative models, namely, TBCNN [12] and ASTNN [25], respectively. TBCNN applies tree-based convolution kernels on an AST to gather the information of child nodes into the parent nodes, and uses dynamic pooling to aggregate the feature vectors of all nodes. The resulted vector is used to represent the entire code snippet. ASTNN splits an AST into a set of subtrees corresponding to the statements in the code snippet. The subtrees are organized into a sequence in accordance with the order of the corresponding statements in the original code. Then it adopts a Bi-GRU network to encode the sequence and uses max pooling to aggregate all hidden states to form the semantic feature vector.

3.2 Enhancing the Discrimination of Semantic Features

Since we grade student code based on the measurement of similarity to the good code, the discrimination of the code feature is critical, i.e., the feature vectors of similar code snippets should be close to each other and the feature vectors of dissimilar code snippets should be far away from each other. To enhance feature discrimination, we design two sub-steps after extracting the semantic features for obtaining the final semantic feature vectors.

Contrastive Learning The first sub-step is to train more discriminative semantic feature vectors using contrastive learning [6]. Contrastive learning represents a category of self-supervised learning methods whose optimization objective is to simultaneously maximize the similarity among positive (close) data points and minimize the similarity among negative (distant) data points. As such it can be used to increase the discrimination of feature vectors. Our design of contrastive learning is as follows. For each code snippet in a random batch, we first construct its positive counterpart based on small random transformation to the source code. We adopt the method in [4] and transform the original source code into a code snippet with equivalent semantic. The transformation has four types including variable renaming, statement swapping, statement insertion and for-while interchanging. Figure 3 shows an example of the transformations, where Figure 3(a) is the original code snippet, and Figure 3(b)∼(e) show the transformed elements in red for all the transformation types. Note that none of the transformations changes the semantic of the original code snippet. For each code snippet, we transform it with the four types and randomly choose a transformed snippet as its positive counterpart during training.

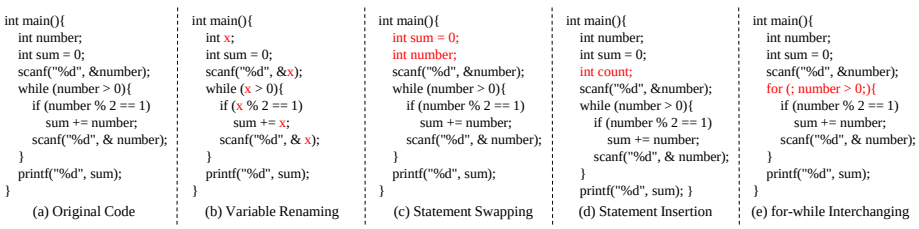


Fig. 3. An example of the code transformation.

The construction of the negative counterpart is relevant to the contrastive loss used for optimization. We experiment with two types of loss functions in the current study, namely, the InfoNCE loss [13] and the Triplet loss [17]. In the former case, for each code snippet, we use all other code snippets in the same batch as the negative counterparts. In the latter case, each code snippet is just paired with one negative counterpart, selected as follows. We first use TBCNN or ASTNN to obtain the encodings of the code snippets. Then for each snippet,

we calculate the L2 distances between its encoding and all other encodings and sort all the distances in either order. We pick the distance in the middle and use the other snippet in the distance calculation as the negative counterpart, as suggested in [17]. Denote by c_i, pos_i and neg_i the i^{th} code snippet in a batch, the corresponding positive counterpart and negative counterpart, respectively. For both types of contrastive loss, the objective function of contrastive learning c_i is depicted as follows:

$$\mathcal{L}^i_{con} = -\log \frac{exp(\frac{e_{c_i} \cdot e_{pos_i}}{\tau})}{exp(\frac{e_{c_i} \cdot e_{pos_i}}{\tau}) + \sum\limits_{neg_i \in batch} exp(\frac{e_{c_i} \cdot e_{neg_i}}{\tau})}, \tag{1}$$

where e_{c_i} denotes the encoding of c_i, $\cdot$ denotes the dot product and τ is the temperature parameter. Note that for the Triplet loss, there is only one neg_i for each c_i and the $\sum$ symbol could be omitted.

Predicting Code Closeness Based on Tree Edit Distance The second sub-step is to further fine-tune the feature vectors obtained in the contrastive learning sub-step using a supervised prediction model. Remember that in contrastive learning, we form a random batch from the student code submitted to all programming questions. As a result, for each code snippet, we equally treat the negative counterparts of the same question and those of a different question. Because the code snippets of the same question are semantically similar, the resulted feature vectors of the same question are still very close to each other. Therefore, we consider to further separate the feature vectors of the code snippets submitted to the same question.

To this end, we train a supervised model to predict the closeness of each pair of code snippets for the same question. We define the closeness of two code snippets as the ratio of the tree edit distance [26] between their ASTs to the number of nodes in the AST with more nodes. Once we obtain the closeness between every pair of code snippets, we set a threshold 0.05 according to the distribution. The pairs with a closeness below the threshold are labeled with 1 (close pairs), and the other pairs are labeled with 0 (non-close pairs). Note that the labels are automatically calculated and do not require any manual labeling effort. Then we train a three-layer fully-connected neural network to predict the labels of the pairs. Each input is a pair of code feature vectors obtained from the contrastive learning sub-step. Because most code pairs are non-close pairs, we use focal loss as the loss function to mitigate the imbalanced distribution problem. Denote by p_i and y_i the predicted probability and label of the i^{th} code pair. The focal loss is depicted as follows:

$$\mathcal{L}^i_{foc} = \begin{cases} -\alpha(1-p_i)^\gamma \log(p_i), & \text{if } y_i = 1 \\ -(1-\alpha)p_i^\gamma \log(1-p_i), & \text{if } y_i = 0, \end{cases} \tag{2}$$

where the α is weighting factor and the γ is the focusing parameter.

At inference time, the output layers of the contrastive learning model and the fine-tuning model are discarded, and the remaining architectures are connected to generate the semantic feature vector for each code snippet.

3.3 Grading Student Code

The above three types of features are concatenated to form the complete feature vector, where each type of feature vector is normalized in the range $[0, 1]$ before concatenation. When grading a code snippet, we first calculate the cosine similarity between its feature vector and the nearest feature vector of a "good" code snippet of the same programming question. Since the similarity score is between -1 and 1, we convert it by adding 1 and then dividing 2 to obtain a score between 0 and 1. To show the grade to the student, we further multiply the score by 100 as the final grade. Denote by v_s and v_g the feature vector of the code snippet and the nearest feature vector of a "good" snippet, respectively. The final grade is calculated as:

$$\text{Grade} = \frac{sim(v_s, v_g) + 1}{2} \times 100, \tag{3}$$

where $sim(v_s, v_g) = v_s^T v_g / \|v_s\| \|v_g\|$ is the cosine similarity between v_s and v_g.

4 Performance Evaluation

We conduct four experiments to determine the model settings, evaluate feature discrimination, evaluate the performance of similarity-based and learning-based grading strategies, respectively. We implement SimGrader using Python 3.7.6 and Pytorch 1.7.0. All source code is available at https://github.com/wangDxia/SimGrader.

4.1 The Datasets and Evaluation Metrics

Datasets We collect 46,949 compilable C code snippets from an OJ system used for an introductory programming course in our university. They are submitted by 146 fresh students in one semester to solve 479 programming questions. We use all the code snippets for feature extraction, contrastive learning and grading.

For the fine-tuning sub-step that predicts code closeness, we extract 73 questions with more than 200 submissions and obtain in total 27,462 code snippets. We form the code pairs for each question and obtain 219000 code pairs, where 41011 pairs are labeled with 1 (close pairs) and 177989 pairs are labeled with 0 (non-close pairs). We randomly divide the pairs with proportion 3:1:1 to form the training, validation and testing set, respectively.

To verify the effectiveness of SimGrader, we construct a small dataset that are labeled by experts. We randomly select 30 questions among the 73 questions with more than 200 submissions, and randomly select 15 submissions for each question. As such we obtain a small dataset with 450 code snippets. We invite two teachers of C programming courses and ask them to independently grade each code snippet on a scale between 1 and 5, which is depicted below. If the two teachers give different grades to a snippet, we ask them to further discuss and make an agreement on the final grade.

- **5 - Correct and graceful:** The code passes all test cases, and the style is clean and clear. There are no redundant variables and code lines, and the variable names are very standardized.
- **4 - Correct with some flaws:** A correct implementation but often accompanied by poor style or complex solution.
- **3 - Nearly correct and neat:** The code does not pass all test cases, but the overall logic is the same as the correct solution, and the code style is clean and clear.
- **2 - Incorrect and confusing:** The code is incorrect and very different from the correct solution. The code style is not good and the logic seems confusing.
- **1 - Incorrect and awful:** The code is incorrect and the code style is awful.

Evaluation Metrics We use multiple metrics to evaluate SimGrader. First, we use the fine-tuning sub-step to determine the best settings for the semantic feature extraction model (i.e., TBCNN or ASTNN) and the contrastive loss (i.e., InfoNCE or Triplet loss). We choose the settings when the model has the best prediction performance on the validation set. The evaluation metric is the accuracy of code closeness prediction.

Second, to evaluate the discrimination of the feature vectors, we cluster the vectors and use two internal metrics to assess the clustering results, namely, Davies-Bouldin Index (DBI) and Silhouette Coefficient (SC). DBI finds for each cluster the most similar cluster based on their diameters, and then computes the average similarity over all the clusters. A smaller DBI value means the clusters are less similar to each other, therefore indicating the vectors are more discriminative. SC is a measure of how similar an object is to the objects within the same cluster compared to the objects outside the cluster. A higher SC value means each object is better matched to the objects inside the same cluster and less matched to the objects outside the cluster, therefore indicating the vectors are more discriminative.

Third, to evaluate the grading performance using similarity measurement, we calculate the correlation between the grades produced by SimGrader and the grades marked by the two experts, on the 450 labeled source code. The used correlation metrics are the Pearson correlation coefficient (PCC) and the Spearman's rank correlation coefficient (SRCC). Furthermore, we use the 450 labeled source code to train and evaluate several supervised models, so that we can compare with existing code grading systems. The evaluation metrics are precision, recall and F1 score.

4.2 The Comparative Methods

We compare SimGrader with with three existing supervised learning methods. The LASSO [18,20] method uses feature engineering and trains a LASSO regression model. The Ensemble [14] method trains an ensemble of feed-forward neural networks on the manually constructed features. The SCG_FBS[16] method trains a Bi-GRU network using the intermediate representation of source code.

4.3 The Hyperparameter Setting

In the contrastive learning sub-step, the output embedding size is set to 64 and the temperature parameter τ is set to 0.1. When training, we set the batch size to 32, the learning rate to 0.001. We use Adamax for optimization. In the fine-tuning sub-step, we set α in the focal loss to 0.25 and γ to 0.98. In the supervised models, the Neural Network model has 4 layers, the number of GBDT's estimators is set to 150, and the SVM kernel is set to linear.

4.4 Experiment 1: Predicting Code Closeness

Table 1 shows the performance of predicting code closeness on the testing set in the fine-tuning sub-step for different model settings. We observe that ASTNN with InfoNCE contrastive learning yields the highest accuracy. In particular, the ASTNN variants perform better than the TBCNN variants. This may be because ASTNN uses the order information of the source code in addition to the AST structural information. The InfoNCE variants perform better than the Triplet variants, which may indicate using more negative counterparts improves the discrimination of the feature vectors. All in all, we use the variant of ASTNN with InfoNCE loss in the subsequent experiments.

Table 1. The performance of different settings for predicting code closeness.

Model	TBCNN		ASTNN	
Variants	InfoNCE	Triplet	InfoNCE	Triplet
Accuracy	0.8265	0.8220	**0.8551**	0.8360

4.5 Experiment 2: Evaluating Feature Discrimination

After the fine-tuning sub-step, the three types of features are concatenated for grading student code. Before we use them for grading, we evaluate their discrimination since the property is critical for similarity measurement. We cluster the 27,462 code snippets of the 73 questions with more than 200 submissions and use DBI and SC to evaluate the clustering performance. We use k-means and set $k = 73$. We compare the results of SimGrader (Full) with the results obtained using the feature vectors produced by the three comparative methods. Also, we remove the contrastive learning (w/o CL) and fine-tuning (w/o FT) sub-step from SimGrader, respectively, and evaluate the features produced by the remaining system. Note that the comparative methods have only a full model, since they do not have a contrastive learning or fine-tuning step as ours. Table 2 shows the results. Remember that a lower DBI and a higher SC indicate the better performance. We observe that SimGrader constantly performs much better than existing methods for both metrics. Moreover, the clustering performance

of SimGrader drops when either sub-step for enhancing feature discrimination is removed. This shows the effect of these two sub-steps. Note that we need at least one of the sub-steps to train the semantic features.

Table 2. The performance of clustering the code feature vectors.

Metrics	DBI			SC		
Model Variants	w/o CL	w/o FT	Full	w/o CL	w/o FT	Full
Ensemble [14]	-	-	1.145	-	-	0.257
SCG_FBS[16]	-	-	1.7718	-	-	0.1644
LASSO [18]	-	-	0.4844	-	-	0.5435
SimGrader	0.3243	0.3233	**0.2948**	0.7059	0.7058	**0.7471**

4.6 Experiment 3: Grading with Similarity Measurement

Our primary contribution is to grade student code based on the measurement of similarity to the good code, so that we don't need large amounts of labeled code and may sufficiently use the massive unlabeled code. To evaluate the accuracy of the grades, we calculate the correlation between the grades produced by SimGrader and the grades marked by the two experts on the 450 code snippets. Table 3 shows the results. We observe that SimGrader constantly performs much better than existing methods for both correlation metrics. Note that for both correlations, a value greater than 0.8 indicates strong correlation. The results indicate the grades produced by SimGrader are very reliable. Moreover, the correlation drops when either contrastive learning or fune-tuning sub-step is removed.

Table 3. The correlation between the grades marked by SimGrader and the experts.

Metrics	PCC			SRCC		
Model Variants	w/o CL	w/o FT	Full	w/o CL	w/o FT	Full
Ensemble [14]	-	-	0.652	-	-	0.645
SCG_FBS[16]	-	-	0.7518	-	-	0.7363
LASSO [18]	-	-	0.8238	-	-	0.8027
SimGrader	0.8701	0.8626	**0.8723**	0.8168	0.7867	**0.8438**

4.7 Experiment 4: Grading with Supervised Learning

Finally, we compare SimGrader with existing supervised learning solutions using the 450 labeled code snippets. We feed the feature vectors produced by

SimGrader into different supervised models and pick the best one for comparison. For the comparative solutions, we also use the best settings reported in the original papers [14,16,18]. Table 4 shows the results. We observe that GBDT performs better than other models. For each model, we observe the performance drop when either contrastive learning or fine-tuning sub-step is removed.

Table 4. Performance of different supervised models using code features extracted by SimGrader.

Models	w/o CL			w/o FT			Full		
	Precision	Recall	F-score	Precision	Recall	F-score	Precision	Recall	F-score
Neural Network	0.7334	0.7388	0.7329	0.7514	0.7455	0.7377	0.7548	0.7466	0.7408
SVM	0.7420	0.7311	0.7293	0.7446	0.7444	0.7415	0.7357	0.7311	0.7320
DecisionTree	0.6963	0.6888	0.6918	0.6649	0.6688	0.6620	0.7005	0.7044	0.7014
RandomForest	0.7936	0.7833	0.7840	0.7238	0.7333	0.7329	0.7843	0.7844	0.7747
GBDT	**0.7970**	**0.7988**	**0.7918**	**0.7677**	**0.7733**	**0.7666**	**0.8197**	**0.8222**	**0.8194**

We use GBDT (Full) to compare with existing supervised learning solutions. Table 5 shows the results. We observe that SimGrader performs much better than comparative methods, which indicates that the features extracted by SimGrader can better capture the static, runtime and semantic property of student code.

Table 5. Comparing SimGrader with existing supervised learning solutions.

	Precision	Recall	F-score
SimGrader(GBDT)	**0.8197**	**0.8222**	**0.8194**
Ensemble [14]	0.5438	0.5511	0.5491
LASSO [18]	0.6285	0.6177	0.6094
SCG_FBS[16]	0.6048	0.5977	0.5827

5 Application: Using SimGrader in An OJ System

We deploy SimGrader in the online judge system used for an introductory C programming course in our university. Originally, whenever a code snippet is submitted for a programming question, the OJ executes it with the pre-defined test cases and gives the feedback in one of the five main types: accepted, wrong answer, time limit exceeded, memory limit exceeded, runtime error. After the deployment of SimGrader, the OJ can in addition give out a grade score to show the overall quality of the code. Figure 4 shows an example, where four code snippets submitted to the same question are graded. The question is to

read three integer values as the side lengths of a triangle and calculate the area of the triangle using Heron's formula.

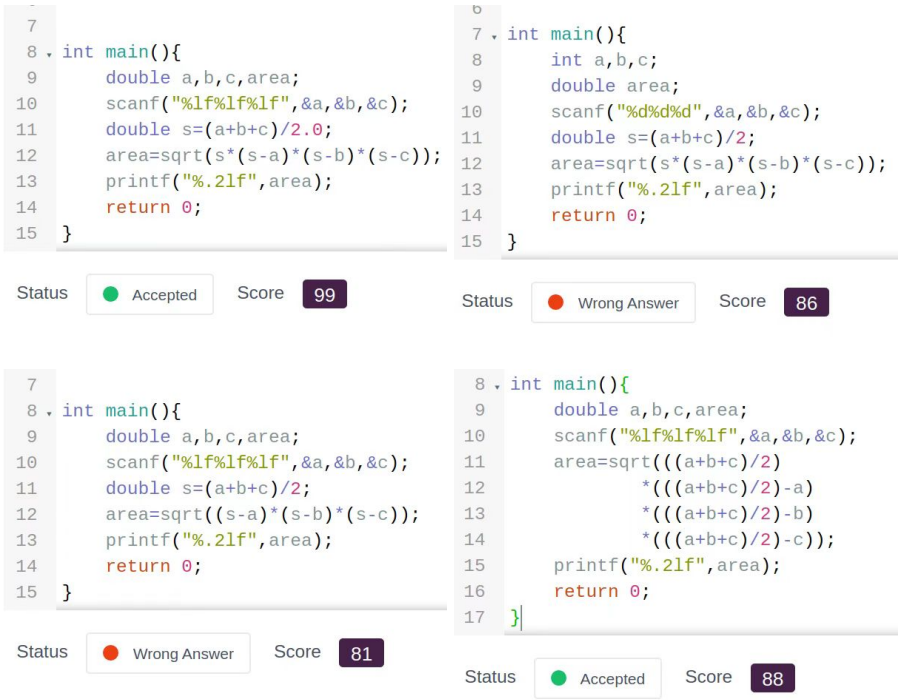


Fig. 4. An example of four code snippets graded by SimGrader.

In the top-left corner, we observe a concise and correct code snippet and SimGrader gives a grade score 99. In the top-right corner, the code snippet is nearly correct except that it fails to convert the integer type into the double type before division (line 9). As such it fails most of the test cases. However, SimGrader finds it very close to the good code and gives a grade score 86. In the bottom-left corner, although the code may pass some cases, it has severe semantic errors. As such SimGrader only grades it as 81. Finally in the bottom-right corner, although the code passes all test cases, SimGrader finds it not concise enough and grades it as 88.

6 Conclusion and Future Work

We design a code grading system, SimGrader, to grade student code based on the measurement of similarity to the good code. As such, we save the expensive overhead to label large amounts of student code required by existing

methods. We extract the static features, runtime features and semantic features to capture the overall quality of each code snippet. To enhance the discrimination of the features, we design the contrastive learning and fine-tuning sub-steps to learn more discriminative semantic features. Finally, the three types of features are concatenated for grading prediction. Experimental results show that SimGrader outperforms existing methods in both unsupervised and supervised learning settings, and justify the effect of each step designed in the system.

The current study shows that the discrimination of the features is critical for code grading based on similarity measurement. As such, we will investigate other methods to produce more discriminative code features in future. Moreover, we plan to collect student code in other programming languages and extend SimGrader to support multiple languages.

Acknowledgement

This work is supported by the grants from the National Natural Science Foundation of China (Grant No. 62137001, 62072185).

References

1. Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations (2018)
2. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019)
3. Bielik, P., Raychev, V., Vechev, M.: Phog: probabilistic model for code. In: International Conference on Machine Learning. pp. 2933–2942. PMLR (2016)
4. Bui, N.D., Yu, Y., Jiang, L.: Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 511–521 (2021)
5. Bui, N.D., Yu, Y., Jiang, L.: Treecaps: Tree-based capsule networks for source code processing. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 30–38 (2021)
6. Chen, T., Kornblith, S., Norouzi, M., Hinton, G.: A simple framework for contrastive learning of visual representations. In: International conference on machine learning. pp. 1597–1607. PMLR (2020)
7. Dong, Y., Hou, J., Lu, X.: An intelligent online judge system for programming training. In: International Conference on Database Systems for Advanced Applications. pp. 785–789. Springer (2020)
8. Hofmeister, J., Siegmund, J., Holt, D.V.: Shorter identifier names take longer to comprehend. In: 2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER). pp. 217–227. IEEE (2017)
9. Johnson-Yu, S., Bowman, N., Sahami, M., Piech, C.: Simgrade: Using code similarity measures for more accurate human grading. In: Proceedings of the 14th International Conference on Educational Data Mining, EDM 2021, virtual, June 29 - July 2, 2021 (2021)

10. Kim, S., Park, J., Jeon, S., Seo, D.: Web-based online judge system for online programming education. In: 2022 IEEE International Conference on Consumer Electronics (ICCE). pp. 1–3. IEEE (2022)
11. Li, Z., Li, L., Wu, Y., Liu, Y., Chen, X.: Automated student code scoring by analyzing grammatical and semantic information of code. In: 2021 16th International Conference on Computer Science & Education (ICCSE). pp. 963–968. IEEE (2021)
12. Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 30 (2016)
13. Van den Oord, A., Li, Y., Vinyals, O.: Representation learning with contrastive predictive coding. arXiv e-prints pp. arXiv–1807 (2018)
14. Orr, J.W., Russell, N.: Automatic assessment of the design quality of python programs with personalized feedback. In: Proceedings of the 14th International Conference on Educational Data Mining, EDM (2021)
15. Peruma, A., Arnaoudova, V., Newman, C.D.: Ideal: An open-source identifier name appraisal tool. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 599–603. IEEE (2021)
16. Qin, Y., Sun, G., Li, J., Hu, T., He, Y.: Scg_fbs: A code grading model for students' program in programming education. In: 2021 13th International Conference on Machine Learning and Computing. pp. 210–216 (2021)
17. Schroff, F., Kalenichenko, D., Philbin, J.: Facenet: A unified embedding for face recognition and clustering. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 815–823 (2015)
18. Singh, G., Srikant, S., Aggarwal, V.: Question independent grading using machine learning: The case of computer program grading. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 263–272 (2016)
19. Srikant, S., Aggarwal, V.: A system to grade computer programming skills using machine learning. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 1887–1896 (2014)
20. Takhar, R., Aggarwal, V.: Grading uncompilable programs. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 9389–9396 (2019)
21. Wang, G.P., Chen, S.Y., Yang, X., Feng, R.: Ojpot: online judge & practice oriented teaching idea in programming courses. European Journal of Engineering Education **41**(3), 304–319 (2016)
22. Wang, M., Han, W., Chen, W.: Metaoj: A massive distributed online judge system. Tsinghua Science and Technology **26**(4), 548–557 (2021)
23. Wasik, S., Antczak, M., Badura, J., Laskowski, A., Sternal, T.: A survey on online judge systems and their applications. ACM Computing Surveys **51**(1), 1–34 (2018)
24. Yang, Z., Yang, D., Dyer, C., He, X., Smola, A., Hovy, E.: Hierarchical attention networks for document classification. In: Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies. pp. 1480–1489 (2016)
25. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 783–794. IEEE (2019)
26. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM journal on computing **18**(6), 1245–1262 (1989)
27. Zhou, W., Pan, Y., Zhou, Y., Sun, G.: The framework of a new online judge system for programming education. In: Proceedings of ACM Turing Celebration Conference-China. pp. 9–14 (2018)