Technological University Dublin

ARROW@TU Dublin

Conference papers

School of Computer Science

2023

Using Machine Learning to Identify Patterns in Learner-Submitted Code for the Purpose of Assessment

Botond Tarcsay

Fernando Perez-Tellez

Jelena Vasic

Follow this and additional works at: https://arrow.tudublin.ie/scschcomcon

Part of the Computer Engineering Commons

This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

Using Machine Learning to Identify Patterns in Learner-Submitted Code for the Purpose of Assessment

Botond Tarcsay(✉), Fernando Perez-Tellez, and Jelena Vasic

School of Enterprise Computing and Digital Transformation, Technological University Dublin, Dublin, Ireland

botond@shoployal.ie, fernandopt@gmail.com, jelena.vasic@tudublin.ie

Abstract. Programming has become an important skill in today's world and is taught widely both in traditional and online settings. Instructors need to grade increasing amounts of student work. Unit testing can contribute to the automation of the grading process but it cannot assess the structure or partial correctness of code, which is needed for finely differentiated grading. This paper builds on previous research that investigated machine learning models for determining the correctness of programs from token-based features of source code and found that some such models can be successful in classifying source code with respect to whether it passes unit tests. This paper makes two further contributions. First, these results are scrutinized under conditions of varying similarity between code instances used for model training and testing, for a better understanding of how well the models generalize. It was found that the models do not generalize outside of groups of code instances performing very similar tasks (corresponding to similar coding assignments). Second, selected binary classification models are used as a base for multi-class prediction with two different methods. Both of these exhibit prediction success well above the random baseline, with potential to contribute to automated assessment with multi-valued measures of quality (grading schemes), in contrast to the binary pass/fail measure associated with unit testing.

Keywords: Applied Machine Learning for Code Assessment · Student Programming Code Grading · Automated Grading

1 Introduction

Manually grading student code submissions on a large scale is a tedious and time-consuming process. With the rise of online programming courses, thousands of submissions must be graded in a short time. Most automated solutions use question-specific unit testing to check if the code is runnable and if it generates the desired output. For some purposes this may be sufficient but, in most contexts, assessing code based on these two properties (execution without error and correct output) is not adequate for different reasons. First, correct output can be generated in different ways. For example, printing

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
A. Y. Rodríguez-González et al. (Eds.): MCPR 2023, LNCS 13902, pp. 47–57, 2023.
https://doi.org/10.1007/978-3-031-33783-3_5

numbers from one to ten can be implemented with a loop or with ten print statements, two approaches that could warrant different grades in spite of appearing the same to a unit test. Second, solutions that do not pass unit tests might still contain sections that go a long way towards solving the given programming problem correctly, or even contain only a typing error that causes the program to fail the unit test.

The aim of our research is to investigate if machine learning methods can be applied to the task of evaluating programs based on source code. Previous work by the authors [2] produced models that successfully learnt, from source code tokens, whether the program consisting of those tokens would pass unit testing or not. The research was limited in scope to code solving simple problems, written by students in the first year of learning to program. It found that both feature sets based on token counts and those based on token sequence information could lead to successful binary prediction (of the unit test pass/fail result), provided that a suitable model type was used with the feature set (Random Forest for token count data and Convolutional Neural Network for token sequence data). The work presented in this paper answers two important further questions about these binary prediction models: (1) How generalizable is the pass/fail prediction success with respect to the task performed by the code? In other words, are the models learning information specific to particular programming problems, from code tackling those particular problems, or are they learning some more general properties of passing vs. failing code? and (2) Can the successful binary classification models be used as the basis for fitting multi-class prediction models, which assign more finely differentiated quality labels (grades)?

Both questions have implications for the eventual application of code-evaluating machine learning models. A more general model allows for more flexible data gathering and training. Hence generalization is an important property to quantify. Also, since multi-class labelled data are not as readily available or abundant as code evaluated through unit testing with the binary pass/fail label, a methodology that makes good use of the latter would be very useful for automating assessment.

This paper is organized as follows: Sect. 2 contains a review of related work, Sect. 3 describes the models previously built by the authors for binary prediction, upon which the work described in this paper is based, Sect. 4 discusses the investigation of generalization (first research question), including methodology and results, Sect. 5, similarly, presents the methodology and results of work on multi-class prediction (second research question) and Sect. 6 contains conclusions, including a discussion of limitations and possible future work.

2 Related Work

A lot of research has been done on understanding source code by different methods and for different purposes. A recent paper [14] reviews the related literature on how automatic grading works and what effect it has on the education system. The author summarized 127 papers on this topic. It was concluded that based on the literature there is no evidence that automated code assessment can or cannot produce as good results as human graders. A comprehensive study [15] explores the current automated grading tools presenting the reason why a new automated assessment model is required. SemCluster [3] uses two

different clustering to group code instances into classes based on their semantic content and to associate them with the relevant programming problem. InferCode [4] looks at code instances individually, using self-supervised machine learning algorithms with natural language processing to deduce the abstract syntax trees of program code by predicting sub-trees. Engine Fuzzer with Montage [13] uses neural network language models to discover vulnerabilities directly in source code. Another important application of source code analysis is plagiarism detection, which is approached with information retrieval techniques in [6] and code structure modelling in [11]. DeepReview [10] utilizes a convolutional neural network for automated code reviews. Another task tackled with the use of deep learning methods is data type prediction for Python: Type4Py [9] uses word2vec embeddings to create type hints based on code context.

Automated grading of student coding work has been a widely researched subject, especially in the last few years due to the increasing demand for programmers and consequent growth in training, as part of both traditional and online courses, where work needs to be evaluated and graded at scale. A paper proposing a method called ProgEdu [7] deals with automated code quality assessment and feedback to students by means of unit testing. A rule-based approach using sophisticated scripts to evaluate, run and grade code is proposed in [5]. AutoGrader [12] compares execution paths using formal methods and reference implementations to evaluate code. A question and language-independent comparison technique based on bag-of-words for distance computation is used in [8], while [16] uses parse tree models and word embedding models to compare the feature vectors created from the source code with LSTMs.

The research presented in this paper takes the approach of treating programs as data in the simplest manner, through the textual tokens of their source code and without intervening representations such as the abstract syntax tree, which is different from what has been published so far in this domain, to the best knowledge of the authors. While the complex structure of most code suggests that this would be a naïve approach, we propose that it is not so in the very specific use case of grading work by early stage programming students, where assignments are elemental. As at the same time it is a use case that entails a large amount of work on the part of instructors, while feedback to students is essential, it is one that would greatly benefit from automation and research towards that goal.

3 Binary Classification

This section gives a brief overview of the binary classification models that provided a starting point for the work discussed in this paper. A detailed account of the work on fitting these models was given in a previous paper by the authors [2].

These models determine whether code will pass unit tests or not, solely based on information derived from the textual tokens in the source code. The availability of large amounts of data for this task, easily labelled through unit testing, makes it a good starting point for the investigation of simple token-based features for evaluating code.

The data used to train the binary classification models was collected by Azcona et al. [1] as part of their research aimed at identifying learning development potential by means of profiling students based on their source code. The data contain half a million source

code instances, each accompanied by the name of the programming problem (set by lecturers) that it attempts to solve and by its unit test pass/fail status (58% fail and 42% pass). These were collected from more than 660 early-stage students of programming, answering 657 different questions in Python across 3 academic years. The dataset is publicly available from the reference link Azcona et al. [1].

Two different types of feature set were designed. In the first type of feature set each feature represents the number of times a particular token appears in a code instance. The number of features in the set is equal to the number of distinct tokens used in all the code in the data set. This type of set we refer to as a token count feature set. In the second type of feature set, the tokens are the features, with their order preserved. In practice, padding is added to bring all code instances to equal length and the tokens are represented by labels. The number of features in this set is equal to the length, in tokens, of the longest code instance. This type we refer to as a token sequence feature set.

Before the feature sets were created, the data set of Python code instances was translated into Python ByteCode, which has the advantage of succinctness in comparison with the Python source code, both in terms of code length and the number of distinct tokens. Both types of feature set were created from each set of code instances (Python and ByteCode) resulting in four feature sets for model fitting.

The token count feature sets were used to fit several conventional machine learning model types, with Decision Tree (F1 = .81, F1 = .85 on Python and ByteCode respectively) and Random Forest (F1 = .83, F1 = .87 on Python and ByteCode respectively) achieving the best prediction scores on test data. The token sequence feature sets were used to fit RNN, LSTM and CNN models. This included additional preprocessing of the data to transform the features into embeddings. The best prediction scores were achieved by CNN (F1 = .76, F1 = .81 with Python and ByteCode respectively). The conclusions of this work were (1) that determining the quality of code based on textual tokens in the source is possible and (2) that ByteCode data can be used for computing-resource efficiency as it results in prediction success comparable to or better than that achieved with Python data.

These promising results warranted further work, with the same feature sets but towards building models for a more granular measure of code quality. The following two sections describe this work.

4 What Did the Binary Classifiers Learn?

While some of the models built on binary (pass/fail)-labelled data displayed very good classification skill, it was still unclear whether the knowledge contained in the models was programming-problem-specific or more general. This is because the models were built with feature sets randomly split into training and testing subsets, with each programming problem likely to be present (via associated code instances) in both subsets.

4.1 Methodology

The first step towards clarifying how general the models are was to define levels of generalization: (1) for any programming problem addressed by a test code instance,

there are training code instances associated with the same programming problem; (2) for any programming problem associated with a test code instance, there are no training code instances associated with the same programming problem but there are training code instances associated with similar programming problems and (3) for any programming problem associated with a test code instance, there are no training code instances associated with the same or similar programming problems.

Next, the best binary classification models were fitted again, with the data splits reflecting the three levels of generalization.

Level 1. Random Forest and CNN models were built for individual programming problems, with both train and test subsets containing code instances associated with one and the same programming problem. The two chosen model types are those that performed best in binary prediction (described in Sect. 3). Random Forest contained 20 estimators with a max depth of 40, while the CNN model used an embedding layer along with a Conv1D layer with GlobalMaxPooling1D and two Dense layers with 25 and 2 neurons.

Three models were built for each programming problem (using 400, 900 and 2000 instances, where available). The purpose of the three models was to understand how the sample size affects the quality of the models.

Level 2. Programming problem similarity was determined based on name semantics. For example, problems named ‘count numbers’, ‘count items’, ‘count up’, ‘count even’, ‘count up to odd’, ‘count down’ and ‘count odd’ were grouped together. Then the code instances associated with the group were segregated by programming problem: in the previous example all instances associated with problems ‘count numbers’, ‘count items’, ‘count up’, ‘count even’ and ‘count up to odd’ were used for model training and instances associated with problems ‘count down’ and ‘count odd’ for testing. As with level 1, CNN and Random Forest were used and models fitted for two different groups of similar programming problems.

Level 3. The programming problem names were used again, but this time to include entirely different problems in the train and test subsets. As in this case the models would have to learn general correct and incorrect code patterns, only sequence-aware model types were used. The LSTM model consisted of an Embedding layer, a SpatialDropout1D layer, 2 LSTM layers with 50 and 25 neurons and a 0.1 dropout, and a Dense layer with 2 neurons. The CNN structure was the same as that for level 1 and level 2 investigations.

4.2 Results

Level 1. Figure 1 shows the F1 scores of models trained on single programming problems. Only programming problems with 2000 instances were used and for each problem three model types were fitted. Thus, for each problem represented by at least 2000 instances in the data set the picture shows three points, indicating (with their y-axis components) the F1 scores for: the problem’s Random Forest model (points with slanted white line), CNN model fitted on ByteCode (points without a line) and CNN model fitted on Python code (points with vertical white line). The Random Forest scores are the

highest but all three model types display scores consistently better than random guessing (.5). Figure 2 shows F1 scores for a representative sample of programming problems, each used to fit CNN models with 400, 900 and 2000 instances. Larger data set sizes generally improve the scores, as expected.

Level 2. Table 1 contains F1 scores for models trained and tested on similar programming problems. Models were built for two groups of similar problems, created as explained in Sect. 4.1. The scores (~.7) testify to a deterioration in the amount of learning achieved, when compared to models built with same problems in the train and test subsets at level 1 (~.8). This indicates that programming-problem-specific patterns are a considerable part of what is learnt by models at level 1.

Level 3. A model that in the context of this research generalizes at level 3 would be learning general code quality indicators, however, as can be seen from Table 2, this is not the case with the feature sets and model types used here. The models that were fitted all exhibit F1 scores only slightly greater than random guessing would produce. This indicates that there could be a very small amount of general learning taking place, but equally that there could be similarities between all the simple programming problems present in the data set. There is a consistent pattern of larger recall for fail than for pass predictions and more similar precision values. This indicates the association of pass labels with specific feature patterns, especially in token sequence data.

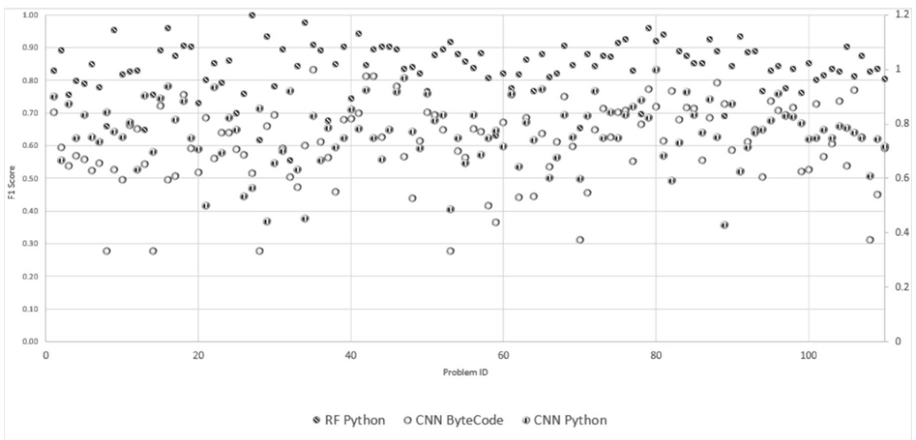

Fig. 1. F1 scores of single-problem models: RF trained on Python token count data and CNN trained on both ByteCode and Python token sequence data (2000 instances in all cases)

Summary. The relative performance of models built with different levels of similarity between the train and test subset in terms of included programming problems shows that the data sets and model types employed here can be trained on source code to tell if unseen simple programs work, but only if taught with samples of code attempting to solve the same or similar problems. This, of course, does not exclude the possibility that further generalization is achievable with more complex feature sets or models.

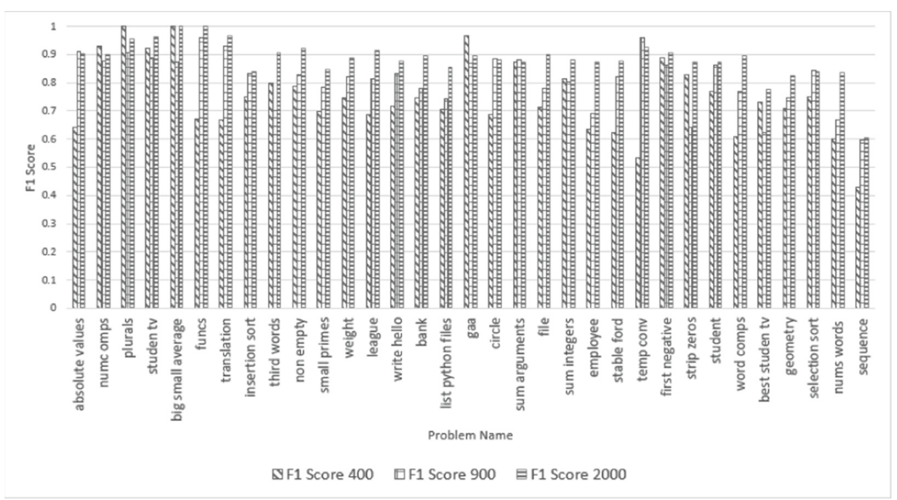

Fig. 2. F1 scores of single-problem CNN models with Python token sequence data, for three different training set sizes (400, 900 and 2000)

Table 1. Performance of models trained and tested on similar programming problems

	Random Forest		CNN	
	ByteCode Token Count	Python Token Count	ByteCode Token Sequence	Python Token Sequence
F1 Score (Group 1)	0.71	0.71	0.69	0.70
F1 Score (Group 2)	0.67	0.68	0.68	0.68

Table 2. Performance of models trained and tested on different programming problems

	ByteCode Token Count		Python Token Count		ByteCode Token Seq.		Python Token Seq.	
	Prec	Recall	Prec	Recall	Prec	Recall	Prec	Recall
	Decision Tree				Long-Short Term Memory Network			
Fail	0.51	0.52	0.53	0.52	0.53	0.72	0.53	0.73
Pass	0.50	0.50	0.53	0.53	0.57	0.37	0.56	0.35
Wgtd F1 Score	0.51		0.53		0.53		0.52	
	Random Forest				Convolutional Neural Network			
Fail	0.52	0.53	0.54	0.55	0.54	0.57	0.52	0.76
Pass	0.52	0.50	0.54	0.48	0.54	0.51	0.55	0.29
Wgtd F1 Score	0.53		0.54		0.54		0.50	

5 Multiclass Prediction

As binary labels are much easier to obtain for code (through unit testing) than fine-grained grades (which typically need to be assigned manually), it would be valuable to have some way of using pass/fail labelled data in the automation of fine-grained grading. Here we look at transfer learning in neural networks as a method for re-using learning with binary-labelled data.

A Random Forest model is also fitted directly with multi-class labelled data.

5.1 Methodology

The data set used for building and testing multiclass prediction models is a subset of 330 code instances from the original data set. All the code instances pertaining to three different programming problems (1530 instances) were manually graded by the authors, with scores between 0 and 10 (6 points were awarded for the presence of various portions of required functionality, 2 for programming style and 2 for correct 'black box' behaviour). The scores were collapsed into ranges corresponding to grades A: 10, B: 8–9, C: 6–7, D: 5–6 and F 0–5. These were further labelled 4(A), 3(B), 2(C), 1(D) and 0(F). To balance out the dataset, the number of instances for each grade was reduced to 66, resulting in a data set of 330 instances. The ratio of the training and testing subsets was 90/10.

Transfer Learning with CNN. To build a multi-class prediction CNN by means of transfer learning, we took advantage of the learning achieved by the CNN Python model fitted on the full data set for binary classification. The network of that model was reused in the network for multi-class prediction, with all weights fixed, except for the top layer, which was replaced in the new network by an output layer with five nodes (instead of two), one for each grade. This network was then trained on 90% of the 330 instances of manually labelled data.

Multi-class Prediction with Random Forest. As Random Forest models were performing well in binary prediction, a multi-class Random Forest model was trained on the 330-instance data set. Transfer learning is not applicable in this case.

5.2 Results

As Table 3 shows, the F1 scores achieved were similar with neural network transfer learning (F = .59 for LSTM and F = .64 for CNN) and the Random Forest model trained from scratch (F1 = .63). Both are better than the expected F1 score of random guessing (.2) for a 5-class prediction task.

Table 3. Performance of multi-class prediction models

	Random Forest Multiclass Python		LSTM Python Transfer Learning		CNN Python Transfer Learning	
	Prec	Recall	Prec	Recall	Prec	Recall
Class A	0.50	1.00	0.67	1.00	0.78	1.00
Class B	0.75	0.75	0.76	0.76	1.00	0.50
Class C	0.40	0.29	0.20	0.17	1.00	0.44
Class D	0.86	0.67	0.67	0.31	0.33	0.43
Class F	1.00	0.50	1.00	0.22	0.50	1.00
Wgtd F1 Score	0.63		0.63		0.64	

Confusion matrix heat maps are shown in Fig. 3 for the CNN and LSTM models and in Fig. 4 for the Random Forest model. From these, it can be seen that the neural network models, using token sequence data, exhibit slightly better performance in terms of closeness of the predicted and real value.

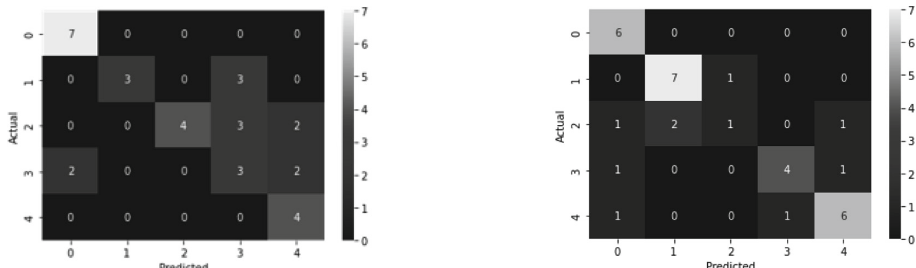

Fig. 3. CNN (left) and LSTM (right) transfer learning confusion matrix heat maps

Summary. The multi-class machine learning prediction models that have been fitted show promise as potential mechanisms for automated grading of programming assignments. An interesting finding is that transfer learning utilizing binary prediction neural network models fitted with a large data set does not seem to produce better results than a Random Forest model fitted with a small amount of multi-class labelled data. However, this work was conducted with code solving only simple programming problems and it is possible that the learning capability harnessed by these two approaches would diverge in the case of more complex code, with the CNN and LSTM performing better in comparison to Random Forest.

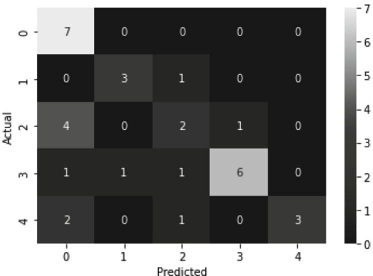

Fig. 4. Random Forest multi-class prediction model confusion matrix heat map

6 Conclusion and Further Work

This paper presented an investigation of machine learning models as a means of evaluating programs on the basis of their source code. Both binary and multiple-class models

resulted in prediction scores better than the baseline. While further work on refining and complementing these models is needed before practical application becomes possible, the value of the work presented here is that it provides evidence of learning in several model types fitted with features derived directly from source code tokens.

Further, it was found that the performance of models that learnt from token counts, in particular Random Forests, and models that learnt from token sequences, in particular CNNs, was not significantly different, indicating that in the case of simple coding tasks the presence of the correct tokens may be sufficient to indicate quality and that the more complex knowledge of sequences may not be required to automatically grade assignments at this level. Further research is needed to explain this, but it can be reasonably assumed that what this is telling us is that if a student knows which constructs to use, then they are probably also going to know in what order to use them. This, of course, would not be applicable in the case of more complex programs, but the findings could be used for the specific purpose of automating the assessment of work by early-stage students of programming.

An admittedly disappointing but valuable finding was that the models do not learn coding patterns in general but rely on being provided correct and incorrect code for a particular problem, or very similar problems, in order to learn how to evaluate code solving that problem. This further narrows but clarifies the possible scope of future application. On the other hand, programming problems posed to beginners are by virtue of their simplicity similar across all instruction contexts, which should allow for pooling of data in the training of a deployed model.

On the whole, the results presented here are promising and establish that it is worth pursuing simple machine learning models as a core component in automated assessment of code written by early-stage programming students. The application envisaged is a service that provides already trained models for common simple programming problems but also employs reinforcement learning to improve performance and build models of code for new problems. Such a service could be offered in variants for instructors and learners, providing constructive feedback in the case of the latter. Also in our future work, we are planning to expand the datasets and include data from other educational institutions to validate this approach as early-stage students for programming seem to follow the same programming patterns independently of the educational institution.

References

1. Azcona, D., Arora, P., Hsiao, I.H., Smeaton, A.: user2code2vec: embeddings for profiling students based on distributional representations of source code. In: Proceedings of the 9th International Conference on Learning Analytics & Knowledge, pp. 86–95. ACM, New York (2019)
2. Tarcsay, B., Vasić, J., Perez-Tellez, F.: Use of machine learning methods in the assessment of programming assignments. In: Sojka, P., Horák, A., Kopeček, I., Pala, K. (eds.) Text, Speech, and Dialogue. TSD 2022. Lecture Notes in Computer Science, vol. 13502, pp. 151–159. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-16270-1_13
3. Perry, D.M., Kim, D., Samanta, R., Zhang, X.: SemCluster: clustering of imperative programming assignments based on quantitative semantic features. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 860–873. ACM, New York (2019)

4. Bui, N.D., Yu, Y., Jiang, L.: InferCode: self-supervised learning of code representations by predicting subtrees. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1186–1197. IEEE (2021)
5. Hegarty-Kelly, E., Mooney, D.A.: Analysis of an automatic grading system within first year computer science programming modules. In Computing Education Practice 2021CEP 2021, pp. 17–20. Association for Computing Machinery, New York (2021)
6. Jayapati, V.S., Venkitaraman, A.: A comparison of information retrieval techniques for detecting source code plagiarism. arXiv preprint arXiv:1902.02407 (2019)
7. Chen, H.M., Chen, W.H., Lee, C.C.: An automated assessment system for analysis of coding convention violations in java programming assignments. J. Inf. Sci. Eng. **34**(5), 1203–1221 (2018)
8. Rai, K.K., Gupta, B., Shokeen, P., Chakraborty, P.: Question independent automated code analysis and grading using bag of words and machine learning. In: 2019 International Conference on Computing, Power and Communication Technologies (GUCON), pp. 93–98. IEEE (2019)
9. Mir, A.M., Latoskinas, E., Proksch, S., Gousios, G.: Type4py: deep similarity learning-based type inference for python. arXiv preprint arXiv:2101.04470 (2021)
10. Li, H.-Y., et al.: Deepreview: automatic code review using deep multi-instance learning. In: Yang, Q., Zhou, Z.-H., Gong, Z., Zhang, M.-L., Huang, S.-J. (eds.) PAKDD 2019. LNCS (LNAI), vol. 11440, pp. 318–330. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16145-3_25
11. Setoodeh, Z., Moosavi, M.R., Fakhrahmad, M., Bidoki, M.: A proposed model for source code reuse detection in computer programs. Iran. J. Sci. Technol. Trans. Electr. Eng. **45**(3), 1001–1014 (2021). https://doi.org/10.1007/s40998-020-00403-8
12. Liu, X., Wang, S., Wang, P., Wu, D.: Automatic grading of programming assignments: an approach based on formal semantics. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), pp. 126–137. IEEE (2019)
13. Lee, S., Han, H., Cha, S.K., Son, S.: Montage: a neural network language model-guided JavaScript engine fuzzer. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 2613–2630 (2020)
14. Combéfis, S.: Automated code assessment for education: review, classification and perspectives on techniques and tools. Software **1**(1), 3–30 (2022)
15. Nayak, S., Agarwal, R., Khatri, S.K.: Automated assessment tools for grading of programming assignments: a review. In: International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 2022, pp. 1–4 (2022)
16. Vimalaraj, H., et al.: Automated programming assignment marking tool. In: IEEE 7th International conference for Convergence in Technology (I2CT), Mumbai, India, 2022, pp. 1–8 (2022)