KING'S College LONDON

King's Research Portal

Document Version
Peer reviewed version

Link to publication record in King's Research Portal

Citation for published version (APA):
Messer, M., Shi, M., Brown, N., & Kölling, M. (in press). Grading Documentation with Machine Learning. In *25th International Conference on Artificial Intelligence in Education* SpringerLink.
https://link.springer.com/chapter/10.1007/978-3-031-64302-6_8

Citing this paper
Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

General rights
Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

•Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
•You may not further distribute the material or use it for any profit-making activity or commercial gain
•You may freely distribute the URL identifying the publication in the Research Portal

Take down policy
If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 08. Oct. 2024

Grading Documentation with Machine Learning

Marcus Messer[1[0000−0001−5915−9153]], Miaojing Shi[2[0000−0002−4933−0073]], Neil C.C. Brown[1[0000−0001−6086−2479]], and Michael Kölling[1[0000−0003−0544−2003]]

[1] Department of Informatics, King's College London, London, UK {marcus.messer, neil.c.c.brown, michael.kolling}@kcl.ac.uk
[2] College of Electronic and Information Engineering, Tongji University, Shanghai, China mshi@tongji.edu.cn

Abstract. Professional developers, and especially students learning to program, often write poor documentation. While automated assessment for programming is becoming more common in educational settings, often using unit tests for code functionality and static analysis for code quality, documentation assessment is typically limited to detecting the presence and the correct formatting of a docstring based on a specified style guide. We aim to investigate how machine learning can be utilised to aid in automating the assessment of documentation quality. We classify a large set of publicly available human-annotated relevance scores between a natural language string and a code string, using traditional approaches, such as Logistic Regression and Random Forest, fine-tuned large language models, such as BERT and GPT, and Low-Rank Adaptation of large language models. Our most accurate mode was a fine-tuned CodeBERT model, resulting in a test accuracy of 89%.

Keywords: Automated Grading · Assessment · Computer Science Education · Machine Learning · Large Language Models · Documentation · Programming Education

1 Introduction

Good documentation is important to any software project [2]. Technical documentation, which includes API reference manuals generated by programs like JavaDoc or PyDoc, offers details on the features and interfaces of a code base [3]. In this paper, we focus on technical documentation, such as JavaDoc or PyDoc, and refer to it simply as documentation. In conjunction with the code itself, the documentation quality affects how maintainable a code base is [1].

The challenge of writing good documentation and assessing documentation quality affects both professional and student programmers. Professional code is usually assessed in a code review. Correctness of the code can be assessed automatically through software testing, but documentation currently cannot be assessed automatically other than trivial checks for formatting. Students need

to learn to write good documentation, but human graders must assess this manually. In addition to the presence and correct style of documentation, good documentation must also relate to the class or function it describes, be understandable, concise and consistent [25].

As student cohort sizes increase in educational settings, automated assessment is becoming more commonplace, with most automated assessment tools using unit tests to assess functionality and static analysis to assess the code [19]. Automatically assessing technical documentation is often limited to checking that documentation is present and matches a specific style guide, using tools such as CheckStyle[3], and does not assess documentation quality [19].

Some assignments manually assess the documentation quality alongside other aspects, including code readability and design. However, manual assessment of documentation quality is time-consuming, especially compared to the near-instantaneous feedback provided by automated assessment tools used to assess a submission's functionality. Automating the assessment of documentation quality could aid the teaching and learning of this important aspect of programming.

In order to automate the assessment of documentation quality, in this paper we will investigate how machine learning techniques can be used to classify how relevant a docstring is to a specific function. The relevance classification could be used as part of a grade for documentation quality alongside static analysis techniques for checking presence and style.

2 Related Work

A recent systematic literature review found only one automated assessment tool between 2017 and 2021 that discussed assessing documentation [19], called AppGrader. AppGrader was developed to provide an automatic assessment of Visual Basic applications, primarily focusing on the functionality of the student's submissions, including validating that specified form objects and common programming concepts are present. In terms of assessing documentation, AppGrader checks to see if functions, conditionals and loops have been comments preceding, on the same line or immediately following the statement. However, AppGrader only validates if the comment is present and not the quality of the content [12].

Validating the presence of documentation is common for many software linters aimed at professional development, including CheckStyle. However, to our knowledge, no professionally developed tools automatically check the documentation quality; thus it is typically part of manual code review.

To relieve developers from writing documentation, software engineering research has utilised neural networks to generate documentation from source code [18, 24, 28]. Similarly, other research has investigated how fine-tuned code-specific large language models can generate documentation and source code [7, 8, 10]. While our task focuses on evaluating the relevance of documentation to the corresponding source code, research into machine learning for code generation and

[3] CheckStyle – A style guide enforcement utility: `https://checkstyle.sourceforge.io/`

summarisation shows that a model can learn the relationship between natural language and code with sufficient data.

Within computer science education, machine learning approaches have been used to assess the quality of a submission's functionality, including assessment criteria that cannot typically be assessed with dynamic or static analysis [20].

One such tool utilised pre-trained image recognition models to assess the image quality of a student's submission in an introductory computer graphics course [21]. Another model was trained to assess the design quality of Python submissions, using abstract syntax trees and a feed-forward neural network to predict a score between 0 and 1 and provide personalised feedback based on features within the individual program [26]. These machine learning-based approaches to automatically assessing elements of programming assignments that are typically manually graded show that machine learning can be adapted for various traditionally manually graded criteria and, therefore, may also apply to documentation relevance.

3 Methodology

3.1 Dataset Selection

We used the CodeSearchNet dataset from GitHub and Microsoft Research [15] to train our machine learning models. The primary dataset consists of two million source code and documentation pairs from many open-source repositories hosted on GitHub, with languages including Python, Java and Go.

Listing 1.1: Example snippet from CodeSearchNet

```
// NL Query: get the description of a http status code
// Relevance Score: 3
private static String toStatusString(int theStatusCode) {
    return Integer.toString(theStatusCode) + "-" +
        defaultString(
            Constants.HTTP_STATUS_NAMES.get(theStatusCode)
        );
}
```

As part of the dataset, which focused on code and documentation[4] generation, the authors of the dataset collected an annotated subset of human-judged relevance scores. The human experts judged a subset of realistic code/documentation pairs. They rated the relevance between the documentation and the code between zero and three inclusive, with zero being irrelevant and three being very relevant. An exemplar snippet of the dataset can be found in Listing 1.1. We filter the subset of code/documentation pairs to include only those written in Java.

[4] In CodeSearchNet, documentation and natural language query are used interchangeably.

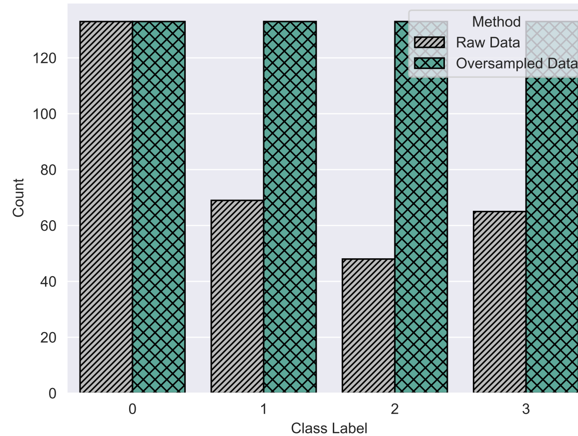

Fig. 1: The distribution of relevance scores before and after oversampling.

While educational programming datasets exist, such as Blackbox [5], FalconCode [11] or data from online judge tools such as HackerRank[5], they often focus on providing a grade for the submission's functionality, and do not contain any data on the documentation quality. CodeSearchNet does not focus on programming education and consists of code and documentation from professionally maintained open-source repositories. However, the large human-expert annotated subset provides an excellent source of ground truth to enable our research into how machine learning can be utilised to grade documentation quality, and to the best of our knowledge, no openly available educational dataset with documentation evaluation exists.

3.2 Pre-Processing

Before training our machine learning models, we pre-processed the CodeSearchNet dataset. While the queries do not meet specific requirements for each language docstring requirements, the natural language queries provide a suitable equivalent for our purposes. We exclude any source code that contains non-ASCII characters to limit our model to classifying source code written in English and to limit issues with vectorisation using pre-trained large language models. Finally, we oversampled the minority classes by sampling with replacement, as the original distribution of classes was heavily skewed; see Figure 1. Oversampling is a common way of balancing classes within a dataset to improve results when training a machine learning model. We chose not to undersample the data, another method for balancing classes within a dataset, as the size of the overall annotated dataset is relatively small.

[5] HackerRank: `https://www.hackerrank.com/`

3.3 Vectorisation

Vectorisation transforms a string of tokens, typically a string of words forming a sentence into a vector, which is then passed to the machine learning model to train or predict a value. There are typically two common approaches to vectorisation: counting the distribution of tokens in a given set of documents or using pre-trained large language models to provide a vector for a specific token.

For counting the distribution of tokens in a given set of documents, we implemented both Bag Of Words and Term Frequency/Inverse Document Frequency (TFIDF), using the implementation provided by Scikit-Learn [22]. Bag of Words is the count of tokens within the training data, with the count of the words being used to generate embeddings to train our models. TFIDF is the proportion of the count of tokens in each string and the inverse of the document count [23].

In recent years, utilising pre-trained large language models has become one of the predominant methods for generating embedding vectors. These pre-trained models use a large dataset; for example, GPT-3 is trained with over 300 billion tokens [6]. The large dataset of pre-trained embeddings often increases accuracy, partially due to the lower likelihood of out-of-vocabulary errors: tokens that do not exist within the training data.

We used HuggingFace [27], which has a large collection of publicly available models, to implement pre-trained models to generate vector embeddings. We chose to use three commonly used pre-trained models, BERT [9], CodeBERT [10] GPT-3 [6]. These models are widely used in various applications from developing chatbots, such as ChatGPT, to code generation and summarization, in tools like GitHub CoPilot[6].

3.4 Model Development

After vectorisation, we trained a number of traditional machine learning models and fine-tuned large language models using 10-fold cross-validation and Optuna [4] for hyperparameter tuning. 10-fold cross-validation splits the training set into ten equal parts, or folds, of shuffled data. Nine of the ten folds are used to train the model, and one fold is withheld to validate the model's accuracy; this process is then repeated so each fold acts as the validation set. The idea of 10-fold cross-validation is to increase the training dataset size by not requiring withholding a large subset of validation data.

For our traditional machine-learning models, we used supervised learning models provided by SciKit-Learn [22], a comprehensive machine-learning library for Python. We trained a number of commonly used traditional machine learning models, including logistic regression, Bernoulli naive Bayes, k-nearest neighbours, random forest, and decision tree classifiers.

We chose to fine-tune two of the three pre-trained large language models we used to generate pre-trained embedding: BERT and CodeBERT, as these are some of the most common ones. We opted against fine-tuning a GPT 3.5

[6] GitHub CoPilot: `https://github.com/features/copilot`

model as it was computationally too intensive. We use the embeddings from the corresponding pre-trained model in the vectorisation stage to fine-tune these models.

We explore how Low-Rank Adaption of Large Language Models (LoRA) can be used to fine-tune existing large language models. LoRA reduces the number of trainable parameters, which decreases the required GPU memory requirement and increases training throughput [14]. Using the same method as fine-tuning the full models, we fine-tune BERT and CodeBERT with LoRA. A summary of our experimentation parameters can be found in Table 1.

Model Type	**Vectorisation Method**			**Hyperparameters**
	BERT	**CodeBERT**	**GPT 3.5**	
Logisitic Regression	✓	✓	✓	
Bernolli naive Bayes	✓	✓	✓	Smoothing: $\{0 - 1\}$
k-Nearest Neighbours	✓	✓	✓	n neighbours: $\{1 - 10\}$
Decision Tree	✓	✓	✓	Max Depth: $\{2 - 20\}$ Min Samples Leaf: $\{5 - 100\}$ Criterion: [Gini, Entropy]
Random Forest	✓	✓	✓	Max Depth: $\{2 - 32\}$x
BERT	✓			Learning Rate: $\{1 \times 10^{-6} - 1 \times 10^{-4}\}$ Batch Size: [16, 32] Epochs: [10, 50, 100]
CodeBERT		✓		Learning Rate: $\{1 \times 10^{-6} - 1 \times 10^{-4}\}$ Batch Size: [16, 32] Epochs: [10, 50, 100]
BERT with LoRA	✓			Learning Rate: $\{1 \times 10^{-6} - 1 \times 10^{-4}\}$ Batch Size: [16, 32] Epochs: [10, 50, 100] LoRA Rank: [8, 16, 32, 64] LoRA Modules: [query, value, key, dense]
CodeBERT with LoRA		✓		Learning Rate: $\{1 \times 10^{-6} - 1 \times 10^{-4}\}$ Batch Size: [16, 32] Epochs: [10, 50, 100] LoRA Rank: [8, 16, 32, 64] LoRA Modules: [query, value, key, dense]

Table 1: A summary of our experiment parameters, all models were trained with the vectorisation methods stated, with Optuna optimising the hyperparameters.

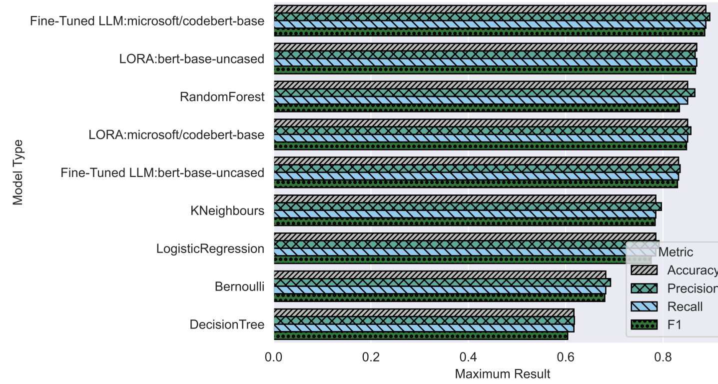

Fig. 2: The maximum results when testing the models against a withheld test set by model type. Precision, Recall and F1 are weighted by the number of true instances for each label.

4 Results

To train our traditional machine learning approaches, we used the traditional methods and the pre-trained large language models to generate the embeddings for the classification. For example, when fine-tuning CodeBERT, we used CodeBERT to tokenize and vectorise the input strings.

Figure 2 shows the maximum accuracy, weighted precision, recall and F1 scores for each of the types of models we trained, with the maximum being calculated after cross-validation and for each set of hyperparameters. Recall is a metric used to quantify the fraction of correctly classified positive instances; accuracy is the ratio of correct predictions over the total number of occurrences; precision is the ratio of correctly predicted positive instances over the total number of positive instances; and F1-Score is the harmonic mean between recall and precision values [13]. As we evaluate a multi-class classification problem, precision, recall and the F1 scores are average weighted by the number of true instances for each label, accounting for the label imbalance within our dataset.

Table 2 shows our top results, including vectorisation, model type and hyperparameters. Our top nine results used CodeBERT for vectorisation and classification with varying hyperparameters. As such, we chose to omit results four through nine, as the results were identical, with an accuracy of 89%. Other models that performed with an accuracy of more than 80% included Random Forest, with BERT embeddings, and a fine-tuned BERT model. Random Forest was the only traditional model to achieve an accuracy higher than 80%, with an accuracy of 87%. All our results can be found in our repository; see Section 7.

Figure 3 shows the total runtime of the training and evaluation of our experiments as reported by Weights and Biases[7], grouped by model type. All models

[7] Weights and Biases: `https://wandb.ai/site`

Rank	Vectorisation Method	Classification Method	Hyperparameters	Acc.	Prec.	Recall	F1
1	CodeBERT	CodeBERT	Epochs: 50 Batch Size: 16 Learning Rate 5×10^{-5}	0.888	0.896	0.888	0.886
2	CodeBERT	CodeBERT	Epochs: 10 Batch Size: 16 Learning Rate 4×10^{-6}	0.888	0.896	0.888	0.886
3	CodeBERT	CodeBERT	Epochs: 50 Batch Size: 32 Learning Rate 7×10^{-5}	0.888	0.896	0.888	0.886
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
10	BERT	LoRA: BERT	Epochs: 10 Batch Size: 16 Learning Rate 1×10^{-5} LoRA Rank: 64 LoRA Modules: [query, value]	0.869	0.868	0.869	0.867
11	BERT	Random Forest	Max Depth = 12	0.850	0.865	0.850	0.834
12	BERT	LoRA: CodeBERT	Epochs: 100 Batch Size: 16 Learning Rate 8×10^{-5} LoRA Rank: 16 LoRA Modules: [value, dense]	0.850	0.856	0.850	0.848

Table 2: The 12 highest performing models, detailing the vectorisation method, classification method, hyperparameters and the performance of the models in terms of accuracy, weighted precision, recall, and F1-score. Results 4 to 9 have been omitted as they are all CodeBERT resulting in the same metric scores with varying hyperparameters.

were trained using a high-performance cluster [16], with 32 CPU cores. LoRA and the fine-tuned approaches were additionally trained on a single Nvidia A100.

5 Discussion

Overall, we showed that machine learning can be used to classify the relevance of natural language describing a corresponding function with an accuracy greater than 85%. While traditional machine learning approaches such as random forest can perform accurately, many traditional approaches fail to achieve an accuracy higher than 80%, with fine-tuned large language models often producing more accurate predictions.

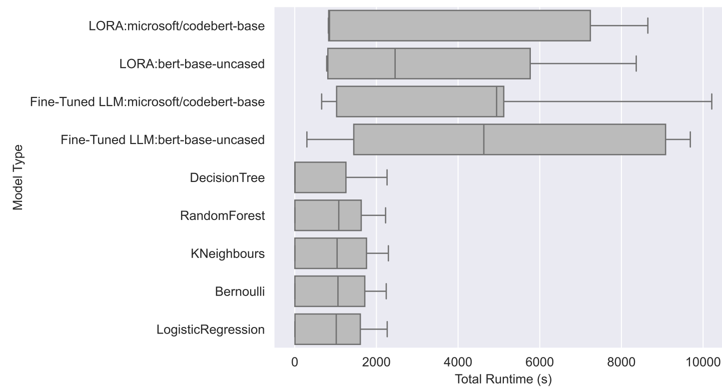

Fig. 3: The total runtime of each model's 10-fold cross-validation training and evaluation, as reported by Weights and Biases, grouped by model type. All approaches were trained on a high-performance cluster [16] with 32 CPU cores, and both the LoRA and Fine-Tuned approaches were additionally trained with a single Nvidia A100.

Our most accurate model was a fine-tuned CodeBERT model with an accuracy of 89%. CodeBERT may have performed better than BERT, as it was trained to generate documentation/code pairs using the CodeSearchNet corpus. Meanwhile, BERT was trained for natural language tasks in English, not code-related.

Our LoRA fine-tuned models performed better than most traditional models, with a LoRA BERT model having an accuracy of 87%. Though the LoRA models underperformed compared to the fully fine-tuned models, the required time to fine-tune and evaluate the model was significantly reduced on average, with the models completing processing approximately 30% faster for BERT and 20% faster for CodeBERT.

As large language models increase in parameter count, fully fine-tuning the parameters becomes less feasible [14], so techniques such as LoRA, which significantly reduce the computing power required to fine-tune an accurate model while sacrificing minimal performance in terms of accuracy, become necessary.

Our results show that a fine-tuned large language model can be used to accurately assign a grade for the relevance of a natural language description of a corresponding function. However, instructors should still validate the models' predictions using their judgement or allow for regrade requests. Regrade requests are common when automating grading using traditional approaches like unit tests or static analysis. The static analysis rules and test cases that instructors write can miss edge cases or not function as intended, providing an incorrect mark for students. While automated assessment tools do not always grade accurately, the benefits of faster feedback and decreased instructor

workload, allowing instructors more time to support students, typically increase student satisfaction.

While our models can provide an accurate grade for the relevance of a documentation/code pair, they do not provide feedback on what was correct or what could be improved. Further research is required to determine how machine-learning approaches could be used to provide feedback to students – whether that is clustering similar submissions, allowing the instructor to provide feedback on a single submission and propagate the feedback to the rest of the cluster [17], or using large language models to generate feedback fully.

5.1 Threats to Validity

Our decision to concatenate the documentation and code strings in the same form as CodeBERT and not use custom processing steps for each large language model may have biased the results to favour CodeBERT. However, BERT is trained exclusively on natural language and not source code, so it has more out-of-vocabulary tokens to generate embeddings for. Out-of-vocabulary tokens are tokens that did not exist in the training data, so the pre-trained large language model cannot provide a known embedding. BERT greedily splits words into subwords, such as 'playing' becomes 'play' and '-ing', and uses these subwords to generate embeddings for out-of-vocabulary words [9].

The bias towards CodeBERT is more prominent when using GPT-3 as a tokenizer, as GPT-3 was trained on a dataset that contained both natural language and source code and was scraped from the internet. While there is a skew towards CodeBERT performing better than other large language models, it does not affect our overall results from this paper, showing that machine learning can be utilised to classify or grade the relevance of documentation of corresponding source code.

We trained our models on documentation and code provided by professionals and not students because, to our knowledge, no openly available education dataset contains annotations for the documentation quality. While the training data is written by professionals and not students in an educational setting, our results show that our methodology can be used to grade the relevance of documentation, and in the future, we aim to adapt our approach to a large educational dataset, similar to the dataset used to evaluate our current models against student submissions.

6 Conclusion

In this paper, we compared five traditional machine-learning approaches, two fine-tuned large language models and two adapter-based fine-tuned large language models to classify the relevance of natural language documentation to its corresponding source code. These documentation/code pairs were classified between zero and three, with zero being irrelevant and three being relevant, and the

ground truth data is provided by human experts as part of the CodeSearchNet corpus.

We train models with various vectorisation methods and perform hyperparameter tuning to train a model with an accuracy higher than 85%. The high accuracy of our model shows that a well-trained model could be used to grade the relevance of documentation to corresponding code. However, this is only a first step towards providing meaningful feedback to students, potentially by generating feedback or clustering similar submissions and propagating feedback from a single graded submission.

6.1 Future Work

In the future, we plan to explore how machine learning can be utilised to grade code quality by replicating our approach for this task and different tasks on an educational dataset we are developing, consisting of grades and feedback for correctness, maintainability, readability and documentation quality. We also plan to investigate how machine learning can be used to generate feedback and to develop a tool that provides grades and feedback for grading documentation quality using machine learning and static analysis.

7 Data Availability

All our raw data, data processing, model training and results can be found on GitHub[8].

Acknowledgements

We thank the King's College Teaching Fund for funding our study and CREATE [16] for providing the high-performance cluster we used to train and evaluate our models.

References

1. Aggarwal, K., Singh, Y., Chhabra, J.: An integrated measure of software maintainability. In: Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318). pp. 235–241 (2002). https://doi.org/10.1109/RAMS.2002.981648
2. Aghajani, E., Nagy, C., Linares-Vásquez, M., et al.: Software documentation: the practitioners' perspective. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. p. 590–601. ICSE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377811.3380405

[8] Data processing repository: `https://github.com/m-messer/Grading-Documentation-with-Machine-Learning`

3. Aghajani, E., Nagy, C., Vega-Márquez, O.L., et al.: Software documentation issues unveiled. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 1199–1210 (2019). https://doi.org/10.1109/ICSE.2019.00122
4. Akiba, T., Sano, S., Yanase, T., et al.: Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. p. 2623–2631. KDD '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3292500.3330701
5. Brown, N.C.C., Kölling, M., McCall, D., et al.: Blackbox: a large scale repository of novice programmers' activity. In: Proceedings of the 45th ACM Technical Symposium on Computer Science Education. p. 223–228. SIGCSE '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2538862.2538924
6. Brown, T., Mann, B., Ryder, N., et al.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper\%5Ffiles/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
7. Chen, M., Tworek, J., Jun, H., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
8. Clement, C.B., Drain, D., Timcheck, J., et al.: Pymt5: multi-mode translation of natural language and python code with transformers. arXiv preprint arXiv:2010.03150 (2020)
9. Devlin, J., Chang, M.W., Lee, K., et al.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2019)
10. Feng, Z., Guo, D., Tang, D., et al.: Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
11. de Freitas, A., Coffman, J., de Freitas, M., et al.: Falconcode: A multiyear dataset of python code samples from an introductory computer science course. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. p. 938–944. SIGCSE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3545945.3569822
12. Gerdes, J.: Developing applications to automatically grade introductory visual basic courses. AMCIS 2017 Proceedings (Aug 2017), https://aisel.aisnet.org/amcis2017/ISEducation/Presentations/28
13. Hossin, M., Sulaiman: A review on evaluation metrics for data classification evaluations. International Journal of Data Mining & Knowledge Management Process (IJDKP) **5** (2015). https://doi.org/10.5121/ijdkp.2015.5201
14. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021)
15. Husain, H., Wu, H.H., Gazit, T., et al.: Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436 (2020)
16. King's College London: King's computational research, engineering and technology environment (CREATE) (2024). https://doi.org/10.18742/rnvf-m076
17. Koivisto, T., Hellas, A.: Evaluating codeclusters for effectively providing feedback on code submissions. In: 2022 IEEE Frontiers in Education Conference (FIE). pp. 1–9 (2022). https://doi.org/10.1109/FIE56618.2022.9962751
18. LeClair, A., Haque, S., Wu, L., et al.: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program

Comprehension. p. 184–195. ICPC ’20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387904.3389268

19. Messer, M., Brown, N.C.C., Kölling, M., et al.: Automated grading and feedback tools for programming education: A systematic review. ACM Trans. Comput. Educ. (Dec 2023). https://doi.org/10.1145/3636515
20. Messer, M., Brown, N.C.C., Kölling, M., et al.: Machine learning-based automated grading and feedback tools for programming: A meta-analysis. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1. p. 491–497. ITiCSE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3587102.3588822
21. Muuli, E., Papli, K., Tõnisson, E., et al.: Automatic assessment of programming assignments using image recognition. In: Data Driven Approaches in Digital Education. pp. 153–163. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66610-5_12
22. Pedregosa, F., Varoquaux, G., Gramfort, A., et al.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
23. Sebastiani, F.: Machine learning in automated text categorization. ACM Comput. Surv. **34**(1), 1–47 (Mar 2002). https://doi.org/10.1145/505282.505283
24. Shi, E., Wang, Y., Du, L., et al.: On the evaluation of neural code summarization. In: Proceedings of the 44th International Conference on Software Engineering. p. 1597–1608. ICSE ’22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510060
25. Treude, C., Middleton, J., Atapattu, T.: Beyond accuracy: assessing software documentation quality. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1509–1512. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3417045
26. Walker, O., Russell, N.: Automatic assessment of the design quality of python programs with personalized feedback. Proceedings of The 14th International Conference on Educational Data Mining pp. 495–501 (2021)
27. Wolf, T., Debut, L., Sanh, V., et al.: Huggingface’s transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2020)
28. Zhang, J., Wang, X., Zhang, H., et al.: Retrieval-based neural source code summarization. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. p. 1385–1397. ICSE ’20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377811.3380383