Machine Learning-Based Automated Grading and Feedback Tools for Programming: A Meta-Analysis

Marcus Messer
Department of Informatics
King's College London
London, UK
marcus.messer@kcl.ac.uk

Neil C. C. Brown
Department of Informatics
King's College London
London, UK
neil.c.c.brown@kcl.ac.uk

Michael Kölling
Department of Informatics
King's College London
London, UK
michael.kolling@kcl.ac.uk

Miaojing Shi
College of Electronic and Information Engineering
Tongji University
Shanghai, China
mshi@tongji.edu.cn

ABSTRACT

Research into automated grading has increased as Computer Science courses grow. Dynamic and static approaches are typically used to implement these graders, the most common implementation being unit testing to grade correctness. This paper expands upon an ongoing systematic literature review to provide an in-depth analysis of how machine learning (ML) has been used to grade and give feedback on programming assignments. We conducted a backward snowball search using the ML papers from an ongoing systematic review and selected 27 papers that met our inclusion criteria. After selecting our papers, we analysed the skills graded, the preprocessing steps, the ML implementation, and the models' evaluations.

We find that most the models are implemented using neural network-based approaches, with most implementing some form of recurrent neural network (RNN), including Long Short-Term Memory, and encoder/decoder with attention mechanisms. Some graders implement traditional ML approaches, typically focused on clustering. Most ML-based automated grading, not many use ML to evaluate maintainability, readability, and documentation, but focus on grading correctness, a problem that dynamic and static analysis techniques, such as unit testing, rule-based program repair, and comparison to models or approved solutions, have mostly resolved. However, some ML-based tools, including those for assessing graphical output, have evaluated the correctness of assignments that conventional implementations cannot.

CCS CONCEPTS

• **General and reference** → *Surveys and overviews*; • **Computing methodologies** → **Machine learning**; • **Applied computing** → *Computer-assisted instruction*; • **Social and professional topics** → **Computing education**.

KEYWORDS

Machine Learning, Automated Grading, Computer Science Education, Meta-Analysis

ACM Reference Format:
Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2023. 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 (ITiCSE 2023), July 8–12, 2023, Turku, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588822

1 INTRODUCTION

As computer science class sizes increase, the number of assignments to grade also increases [21], resulting in further research into automated grading and feedback. The use of automated graders for programming tasks originated in the 1960s [19]. Since then, research has primarily focused on grading the correctness of students' source code using dynamic and static analysis techniques [3, 28].

We define four criteria for grading programming assignments:

Correctness: Typically can be the syntactic correctness (can the program compile), the correctness of the implementation (has the student implemented the intended functionality?) or the correctness of methodology (has the student implemented the specific method or language feature the instructor has requested?).

Maintainability: Evaluates the effectiveness of a student's implementation of code that is easily extensible in the future. This could include designing a code base to minimise coupling and maximise cohesion, whether the student used polymorphism and inheritance correctly, or whether the student used functions to reduce duplicated code.

Readability: Assesses how easy to comprehend a student's submission is. While maintainable code can make the code more understandable, other source code characteristics can also indicate whether the code is readable. Including following code style principles, naming classes, methods, and variables in meaningful ways, substituting constants for magic numbers, and indenting code at the correct levels.

Documentation: Gauges whether a student's docstrings and inline comments are sufficient to explain their code to other programmers. This typically includes validating that the documentation exists, relates to the code it is explaining and is not overly commented on with comments that state the obvious.

There is some discussion of when maintainability, readability and documentation should be included when grading novice programming assignments, whether that is grading these skills in CS1/CS2 classes or in later courses such as Software Engineering. While this discussion is important and requires further research, this article includes auto-graders from all levels computer science courses, from use of block-based languages to grading of maintainability in a 200-level course [10].

This article builds upon an ongoing systematic literature review on automated grading and feedback tools for programming assignments. In this meta-analysis, we focus on machine learning (ML) auto-graders. The implementation of traditional auto-graders is well-researched; most auto-graders use a dynamic or static technique [3, 28]. However, less research has been conducted on ML-based auto-grading of programming tasks. As a result, we intend to analyse and contrast the various implementations of ML automatic graders and feedback systems in-depth.

This meta-analysis provides an in-depth analysis of the role of ML in automated grading and feedback of programming assignments by contributing the following:

- A focused review of machine learning implementations of automated graders and feedback tools.
- Detailed statistics of machine learning paradigms, categories, and preprocessing and evaluation techniques.
- An in-depth discussion on the implementations and the gaps and drawbacks of the current research.

The article is organised as follows: The differences between this article and related work are discussed in section 2; our methodology and inclusion and exclusion criteria are outlined in section 3. Section 4 details the articles we investigated and the outcomes of our search, while section 5 discusses what we observed. Section 6 finalises what we discovered and outlines future work.

2 RELATED WORK

Most systematic reviews of automated grading focus on collating and annotating grading or feedback approaches. Ala-Mutka [3] categorises automatic grading tools into two types, dynamic and static, and discusses the benefits and drawbacks of automated assessment. They conclude that the formally defined source code structures are amenable to automated assessment and that careful consideration should be used when incorporating them into education.

While Ala-Mutka investigates grading approaches, Keuning et al. [20]'s review investigates the automated feedback tools. They define a set of feedback categories, which include knowledge about task constraints and knowledge of mistakes. Using these categories, they annotate their included papers and conclude that most automated feedback tools offer feedback on solution errors, and that very few give feedback on how to proceed.

Paiva et al. [28]'s systematic literature review focused on classifying which computer science domains were automatically assessed and evaluated feedback using Keuning et al. [20]'s feedback categories. Their review concluded that researchers had tried to automate the assessment of most practical tasks of the CS curriculum and that various methods are used to provide feedback, including automated program repair and source code metrics. Additionally, they determine that most tools are developed as prototypes for a specific research study or course and receive no further development or wide-scale deployment.

Our ongoing systematic review investigates which programming skills automated grading and feedback tools evaluate and which techniques are used. Furthermore, we investigate which language paradigms are auto-graded and how the tools are evaluated.

Other reviews have investigated how ML can be utilised in automated grading outside the computer science domain. Zhai et al. [40] conducted a systematic review of applying ML in natural science assessments. They concluded that most studies used supervised ML approaches, and approximately half embedded ML directly into the learning environment. Systematic literature reviews were also conducted on prose-based assessments, including short-answer questions [13] and essays [30]. Both reviews found that neural network-based approaches were commonly used to assess short-answer and essay questions.

Our meta-analysis builds on an ongoing systematic literature review by emphasising the use of ML to automatically grade or provide feedback on programming assignments – something that, to our knowledge, has not been covered by any existing reviews.

3 METHODOLOGY

To investigate how ML has been implemented in automated graders and feedback tools, we aim to answer the following research question:

RQ1 Which machine learning auto-graders grade fundamental programming skills?
RQ2 Which techniques are implemented by machine learning auto-graders to predict a grade or generate feedback?
RQ3 Which criteria are used to assess machine learning auto-graders, and how do they perform?

As part of our ongoing systematic literature review, we used two separate search strings in ACM DL, IEEEXplore, and Scopus, one for automated grading (Listing 1) and the other for automated feedback (Listing 2). We chose these databases since ACM or IEEE publishes most Computer Science Education resources. We used Scopus, the world's largest abstract and citation database for peer-reviewed literature, to find additional sources outside of ACM and IEEE. Our search strings were transformed into each database's specific format to verify that the search behaviour was consistent. We then conducted multiple screening stages to determine if the articles found should be included in the review.

While completing the final stages of the systematic literature review, we conducted a backward snowball search [38] using the 14 ML-based papers from the ongoing systematic review. Using these papers as a basis, we investigated which papers they had cited to find any other research potentially involving ML applied to the grading or giving feedback on programming assignments. After finding potential papers, we analysed the title and abstracts to determine if they should be included or excluded, using the criteria found in Table 1. Searching for papers' references and screening

Inclusion	Exclusion
The paper is a primary source.	The paper is not written in English.
The paper focuses on auto-grading or feedback on source code.	The paper is not a research study or peer-reviewed.
The tool supports a textual or block-based programming language.	The paper is not accessible via university subscriptions.
Papers published from 2012 to 2021 inclusive.	Papers that do not contain sufficient detail on the machine learning implementation.

Table 1: The inclusion and exclusion criteria

them against our exclusion criteria was repeated twice, using the papers from the previous iteration. This resulted in a backward snowball search with a depth of three.

Our ongoing systematic review limited our search to papers published from 2017 to 2021 inclusive. In this meta-analysis, we expanded our search dates to those published between 2012 and 2021 inclusive. Additionally, we decided to include block-based tools in addition to tools that grade textual programming languages and exclude papers that did not provide sufficient detail on their ML implementation. We included block-based tools and expanded our search dates to increase the proportion of ML papers in the analysis. While block-based tools are significantly different from textual programming languages and the learning aims of these approaches can differ, some block-based ML approaches could be adapted to textual programming languages.

After completing our initial search for relevant cited papers, we repeated our backward snowball search approach twice for the papers we found and included at each level. We then conducted a full-text screening and extracted data.

To answer our research questions, we extracted and analysed what core skills the tools graded, their preprocessing steps, the ML approach, and how the tools were evaluated [1]. Additionally, we grouped specific ML models into overarching categories. The first is traditional approaches, including statistical-based models like logistic regression, and the second is neural network-based models, including convolutional and recurrent neural networks.

```
(programming OR source code) AND
(grade OR grading OR grader OR
    mark OR marks OR marking) AND
(assignment OR exercise OR assessment OR course) AND
NOT(robot* OR vulnerability OR ICT)
```

Listing 1: Grading Search String

```
(programming OR source code OR student code) AND
(feedback OR hint) AND
(assignment OR exercise OR
    submission OR novice programm*) AND
NOT(robot* OR vulnerability OR ICT)
```

Listing 2: Feedback Search String

[1]Raw data and data processing repository: https://github.com/m-messer/In-Depth_Analysis_ML_Graders_Analysis

4 RESULTS

From the initial 14 papers, our backward snowball search resulted in the following:

Stage 1 52 titles and abstracts screened; 11 included in stage 2.
Stage 2 12 titles and abstracts screened; 1 included in stage 3.
Stage 3 1 included paper.

The 14 initial papers from the systematic review and the 13 from the backward search resulted in 27 papers to screen during the full-text review. Nine were excluded, leaving 18 articles from which to gather data.

Most papers were excluded during the snowball search's title and abstract screening because they did not focus on ML, while some were excluded because they were duplicates or did not grade source code. Whereas, during the full-text review, most were excluded as they did not provide sufficient detail on their ML implementation.

In Figure 1, we present the trend in ML implementations in automated grading and feedback tools. Surprisingly, the trend does not follow the same upward trend as auto-graders [28]; instead, it has varying publications per year. This variation might be due to our exclusion criteria; some papers use ML methods within the education domain but do not specify their implementation in detail [9, 11, 26]. They may not include their specific implementation in their paper as they discuss the pedagogical effect [9], or implement commercially available approaches [26].

We previously introduced the fundamental programming skills of correctness, maintainability, readability and documentation; Table 2 shows the papers that grade these skills and the ML categories used. Furthermore, Table 2 shows that most ML-based tools grade

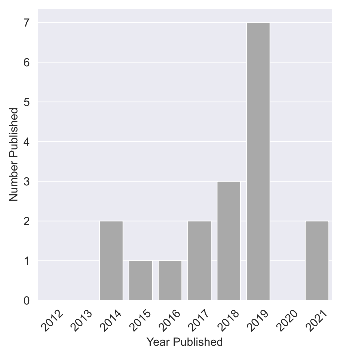

Figure 1: The number of found publications per year.

or give feedback on the correctness (56%) or syntactic correctness (33%), a combination of correctness and readability (6%) or correctness and maintainability (6%). However, none of these tools uses ML to grade documentation. Table 2 also shows that most tools have implemented a neural network-based approach (61%) or a traditional approach (33%).

Figure 2 shows the proportion of occurrences for each preprocessing technique; each paper implemented one or more of these techniques. Most implementations normalize the tokens, such as converting string or number literals or variables to a specific token [1, 15–17, 27, 32, 34, 36]. Another popular method is to convert the source code to a graph, mostly an abstract syntax tree (AST) [14, 16, 22, 24, 29, 35, 36]. However, one article additionally converts the source code to control-flow (CFG), and data-dependency graphs (DDG) [35]. Other preprocessing steps include building a good set of submissions using unit tests [22, 33, 35], removing comments [27, 34], and replacing rarely used tokens with a general token [5].

The included papers utilised one or more common ML paradigms. The most commonly used paradigms used by the tools, were approaches based on the availability of ground truth data [31]:

- Supervised learning is when a model is trained using some training set of labelled data and learns a function to map the features to the label.
- Unsupervised learning models do not require a set of labelled data; instead, learning through patterns within the data.
- Semi-supervised learning is when the model has a subset of the overall training data labelled, enabling a model to learn by combining supervised and unsupervised approaches.

Figure 3 shows that 74% of our included papers utilised a supervised learning technique and 16% opting for an unsupervised learning approach. Gupta et al. [15] implemented two models, one that used a semi-supervised Long Short-Term Memory model, and the other using a reinforcement learning model. A reinforcement learning is when the model or agent learns from a series of reinforcements, either rewards or punishments [31].

We categorised the models into traditional and neural network-based approaches to investigate the overall trend in the models that ML-based tools trained to award grades and give feedback. Traditional approaches include models such as ridge regression [35], random forest [22] and support vector machines [36]. Neural network-based approaches include models such as convolutional neural networks (CNNs) [16, 34], recurrent neural networks (RNNs) [5], Long Short-Term Memory (LSTM) models [15, 27, 32], and encoder/decoder models [1, 29].

		ML Category		
		Neural Network-Based	Traditional	Both
Skill	Correctness	[10, 16, 27, 29, 34, 39]	[14, 22, 24, 33, 35, 36]	
	Syntactic Correctness	[1, 2, 5, 15, 17]		[32]
	Maintainability	[10]		
	Readability	[29]		
	Documentation			

Table 2: The resulting papers, categorised by skill graded and ML Category. Some papers grade more than one skill.

In Figure 4, we show the evaluation techniques used by the included articles to evaluate their models; some papers included multiple evaluation techniques. Most models are evaluated on how accurately they can predict the correct grade or feedback based on the test data, typically a subset of the original training data [2, 5, 15–17, 22, 27, 29, 32, 34, 39]. Other standard ML metrics have also been used, including precision and recall [2, 24, 29], mean absolute error [32, 33, 36], and Pearson correlation coefficient [33, 35]. Furthermore, some papers opted to compare their models against other baseline datasets or models [1, 5, 15, 16, 29, 39] or compared to datasets manually annotated by a human [14, 36, 39]. Additionally, some implemented cross-validation to validate the stability of their ML models [33–36], and others chose to perform a case study to verify how well their tool works when given to students [2, 10, 35].

5 DISCUSSION

5.1 Skills Graded

5.1.1 Correctness. As shown in our results, all ML-based tools we have analysed focused on automating the grading or feedback of correctness, either syntactic or functional. Researchers may focus on grading functional correctness, as one of the aspiring software engineers' primary tasks is creating software that meets the desired requirements. A more straightforward and prominent task for novice software engineers is creating code that can compile, hence the focus on syntactic correctness. None of the analysed papers investigated grading the correctness of methodology. This could be due to static analysis techniques being easier to implement and more accurate than ML for detecting specific language techniques.

5.1.2 Maintainability. Only one of our analysed papers evaluated maintainability: Day et al. [10] implemented a multi-layer perceptron neural network to generate feedback on both correctness and maintainability for their Eclipse IDE plug-in. They used a supervised learning approach, using data extracted from a student's code and events that are detected with the IDE. Though they used a neural network to generate feedback, the evaluation of correctness used dynamic and static approaches, such as unit tests and error types. Similarly, maintainability was evaluated using source code metrics, a static analysis approach, including the total lines of code and cyclomatic complexity.

Using a static approach, especially metrics, is a common way to evaluate maintainability; many of these use data from source code converted to a graph to calculate a value for evaluation, such as cyclomatic complexity [25] and depth of inheritance tree [7]. While using graph structures is a typical approach to evaluate maintainability, the analysed papers use these graphs to grade correctness [14, 16, 22, 24, 29, 35, 36]. Further research could be conducted into using these graphs and an ML approach to grade maintainability, as both techniques have been implemented to evaluate source code.

5.1.3 Readability. Similarly to maintainability, only one of our included papers evaluated readability: Piech et al. [29] implemented an encoder/decoder-based neural network to learn programming

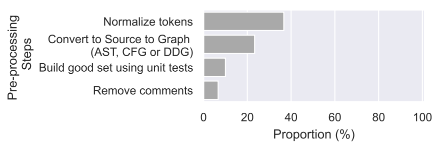

Figure 2 The percentage of preprocessing methods. Results with only one occurrence are omitted.

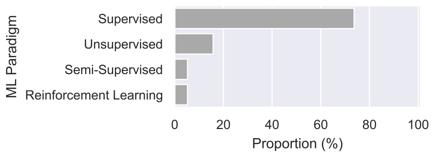

Figure 3 The proportion of primary ML paradigm. Some articles implemented multiple paradigms.

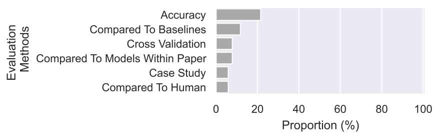

Figure 4 The percentage of evaluation techniques. Results with only one occurrence are omitted.

embeddings to propagate feedback on both correctness and readability.

They utilise *Hoare triples* [18], tuples containing a precondition, an executable program, and a postcondition, as the basis of their dataset. They implement models to encode the precondition to a m-dimensional feature representation, transform the encoded precondition and executable program to an encoded postcondition, and a final model to decode the postcondition. Once the encoder/decoder model has been trained and optimised, they utilise the embeddings, instructor-annotated feedback, and data from ASTs to recapture program structure and style elements to train an RNN to propagate feedback on both correctness and maintainability.

While Piech et al. [29] utilised ASTs in their model to recapture structure and stylistic elements, other ML research has implemented models to learn natural coding conventions [4], evaluate method name consistency [23], and evaluate semantic representations of identifier names [37], of professional source code. These approaches could be adapted and expanded to provide grades and feedback on the readability of novice code.

5.1.4 Documentation. While correctness, maintainability and readability had at least one paper focused on them, documentation was not evaluated by any of the included articles.

While there is little research into grading documentation, generating code from the documentation and generating documentation from code is currently heavily researched. GitHub CoPilot[2] is one such tool that generates code from the documentation. They utilise a GPT language model called Codex [6] to generate the source code, while other language models, such as CodeBERT, have been utilised to generate documentation [12]. PyMT5 is another language model used to generate source code from documentation and documentation from the source code [8]. These three approaches utilise a fine-tuned bimodal model, specifically using existing documentation paired with source code snippets to train their models, which could be utilised to grade or give feedback on documentation.

5.2 Paradigms Implemented

Most of the included papers have trained their model using a supervised method, less than 20% implemented an unsupervised model, and only one article implemented a semi-supervised or reinforcement learning approach. Gupta et al. [15] implemented a deep reinforcement learning model to offer feedback on syntactic correctness. They opted to use the asynchronous advantage actor-critic (A3C) algorithm, which utilises multiple actor-critic threads in parallel to stabilise the learning process. The actor-critic thread is an example of the policy gradient method, which learns both the actor and the critic, where the critic evaluates how advantageous it is to be a new state.

They define the A3C algorithm to repair source code, where the agent can either perform navigate actions, editing the state of a cursor, or edit actions, editing the state of a string. Using these actions, an agent attempts to repair the program and is fully rewarded if the code compiles or partially rewarded if the agent has fixed at least one error. Using their A3C algorithm, they train an LSTM to embed the tokenized code into a vector, which is then used to learn the policy and value functions. To accelerate the training of their model, they guided their agent with expert demonstrations of how to repair the source code, thus implementing a pseudo-semi-supervised reinforcement learning approach.

While Gupta et al. implemented a semi-supervised approach for their reinforcement model, none of the included papers implemented a typical semi-supervised learning model, such as label propagation or general adversarial networks. Although semi-supervised learning reduces the effort required to manually label an entire dataset, allowing models to be trained using a small subset of labelled data, they can be more challenging to implement.

Although supervised learning requires a large set of labelled data, which can be time-consuming to annotate, it can produce better results and be simpler to implement. The supervised model can provide better results, as the model has example relationships between the source code and the grade or feedback. While supervised learning can often provide better results, recent research using unsupervised learning in other software engineering domains has produced good results, including GitHub CoPilot's underlying model Codex, which fine-tunes GPT-3, uses an unsupervised approach to learn a language model [6].

5.3 Categories Implemented

While most of the included papers focus on implementing traditional or neural network-based approaches, Santos et al. [32] implemented an n-gram model and an LSTM to train language models to provide feedback on syntactic correctness. They implemented a 10-gram model, where 10 contiguous tokens will be grouped into a 10-gram, with each token being delimited by a space. They then split the source code into 21 token windows, which are compared against the model for their cross-entropy. The cross-entropy is then converted to a specific value for each token, and the token with the highest value is considered the error. Finally, the n-gram model

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

attempted to delete or insert a token before or substitute the token to repair the source code.

For their LSTM model, they converted each source file into a vector and then converted it to a one-hot encoded matrix. In a one-hot encoding, precisely one item in each column has the value one; the rest of the values in the column are zero. Using this one-hot encoding, they train an LSTM to map the contexts to categorical distributions of adjacent tokens. The final model was then used to find the likely error location and suggest an edit.

When comparing which method can suggest a true fix as its first suggestion, the abstract 10-gram model, which only requires the correct operation, location and token type, outperforms the 10-gram concrete model, which additionally requires the fix to produce the exact identifier or literal, and the LSTM, even when the hyperparameters are fine-tuned.

Even though the traditional approach gave better results than the neural network-based technique in this instance, many of the included articles opted for the neural network-based approach. The prevalence of neural network-based techniques as the primary method could be due to how successful these techniques have been when applied to source code in domains other than computer science education. Some tools, such as GitHub CoPilot, developed for other software engineering domains, have become popular tools to aid developers. Thus, authors may have adapted the ideas of using these neural network-based language models to apply to the computer science education domain.

5.4 Evaluation

5.4.1 Techniques. Most of our included papers opted to evaluate their model's accuracy, confirming how well they predict the desired grade or feedback. However, few compare the model's predictions with a human grader manually labelling the same dataset, even if the data used was taken from internal assignments that would typically be manually graded. Although some articles evaluated their models against internal datasets, others used open datasets, such as GitHub [27, 32], Blackbox [32], and code.org [29, 39]. While comparing to a dataset graded by humans offers a realistic baseline, manual annotation requires a substantial time commitment, may not always yield accurate results and requires ethical approval to use students' actual grades.

Furthermore, few papers conducted a case study to verify how well their tool performs in a real-world scenario with actual end-users, both students and instructors. Validating the tool's performance in a real-world setting can aid researchers by receiving feedback on the quality of the grades or feedback given to students, the overall experience of using the tool, or how to improve the functionality of the tool.

5.4.2 Performance. Typically, the implemented models performed well, with those using accuracy metrics predicting the correct grade or feedback with at least 70%. Some papers report an accuracy of approximately 95% for specific problems within their dataset. For those articles that compared their models to other baseline models, they typically improved the performance. Those articles that ran case studies generally saw good results, either by helping the students resolve errors faster or performing better than unit test-based tools.

6 CONCLUSION

In this meta-analysis, we categorised the included approaches based on the fundamental programming skills graded, the ML paradigm and the categories implemented and investigated their evaluation techniques and how they performed.

We discovered that most ML-based tools were primarily concerned with functional or syntactic correctness. Few papers proposed methods to grade maintainability or readability and no articles that scored documentation. It seems like a squandered opportunity to grade correctness using ML rather than maintainability, readability, or documentation, as correctness grading using dynamic or static techniques is widely covered. However, auto-grading correctness using ML could reduce the effort by instructors, as they would not have to implement a comprehensive test suite, and the ML approaches typically do not need to be revised when a new version of the assignment is released.

Most of the included papers implemented a supervised model, with most using a neural network-based model and fewer using traditional models. Few implemented unsupervised techniques, and only one paper opted for a deep reinforcement learning approach. While supervised approaches can provide better results, they require a large set of labelled data, which can be time-consuming to annotate. Additionally, unsupervised approaches have seen recent success in other software engineering domains, such as code generation.

The evaluation of these models primarily focused on their predictions' accuracy, in which most models performed with an accuracy of at least 70%, improving the accuracy compared to baseline models and significantly improving on classical approaches, such as unit testing. Additionally, some papers compared the models to a human-graded benchmark, while others performed a case study to validate their performance in a real-world scenario.

6.1 Future Work

Most of the current research is into applying ML to automated grading and feedback, and potential future work could include producing ML-based tools to grade readability, maintainability and documentation. We have nearly concluded our systematic literature review and are in the final drafting stage before submission to a journal. In the future, we plan to extend this work by conducting a forward snowball search of the papers included in this article and by extending our search to include commercially available tools, including large language models, such as Codex and GPT-4. Thus analysing recent developments in applying ML to automated grading and feedback and how ML is applied to accelerate professional software engineering. Furthermore, we plan to evaluate these approaches against a shared dataset to compare their performance.

REFERENCES

[1] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: For the student programs, from the student programs. *Proceedings - International Conference on Software Engineering* 18 (5 2018), 78–87. https://doi.org/10.1145/3183377.3183383

[2] Umair Z. Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. 2019. Targeted example generation for compilation errors. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019* 34 (11 2019), 327–338. https://doi.org/10.1109/ASE.2019.00039

[3] Kirsti M. Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer science education* 15, 2 (2005), 83–102.

https://doi.org/10.1080/08993400500150747

[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 281–293. https://doi.org/10.1145/2635868.2635883

[5] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. *Proceedings - International Conference on Software Engineering* 2018-January (2018), 60–70. https://doi.org/10.1145/3180155.3180219

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021. Evaluating Large Language Models Trained on Code.

[7] Shyam R. Chidamber and Chris F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20 (1994), 476–493. Issue 6. https://doi.org/10.1109/32.295895

[8] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. , 9052–9065 pages.

[9] Gilbert Cruz, Jacob Jones, Meagan Morrow, Andres Gonzalez, and Bruce Gooch. 2017. An AI system for coaching novice programmers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10296 LNCS (2017), 12–21. https://doi.org/10.1007/978-3-319-58515-4_2/FIGURES/3

[10] Melissa Day, Manohara Rao Penumala, and Javier Gonzalez-Sanchez. 2019. Annete: An Intelligent Tutoring Companion Embedded into the Eclipse IDE. In *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*. IEEE, New York, NY, USA, 71–80. https://doi.org/10.1109/CogMI48466.2019.00018

[11] Yu Dong, Jingyang Hou, and Xuesong Lu. 2020. An Intelligent Online Judge System for Programming Training. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12114 LNCS (2020), 785–789. https://doi.org/10.1007/978-3-030-59419-0_57/FIGURES/3

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. https://doi.org/10.48550/ARXIV.2002.08155

[13] Lucas Busatta Galhardi and Jacques Duílio Brancher. 2018. Machine learning approach for automatic short answer grading: A systematic review. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11238 LNAI (2018), 380–391. https://doi.org/10.1007/978-3-030-03928-8_31/TABLES/3

[14] Elena L. Glassman, Rishabh Singh, and Robert C. Miller. 2014. Feature Engineering for Clustering Student Solutions. In *Proceedings of the First ACM Conference on Learning Scale Conference* (Atlanta, Georgia, USA) *(LS '14)*. Association for Computing Machinery, New York, NY, USA, 171–172. https://doi.org/10.1145/2556325.2567865

[15] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (7 2019), 930–937. Issue 01. https://doi.org/10.1609/AAAI.V33I01.3301930

[16] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural Attribution for Semantic Bug-Localization in Student Programs. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc.

[17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 31 (2 2017), 1345–1351. Issue 1. https://doi.org/10.1609/AAAI.V31I1.10742

[18] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

[19] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3 (10 1960), 528–529. Issue 10. https://doi.org/10.1145/367415.367422

[20] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education (TOCE)* 19 (9 2018). Issue 1. https://doi.org/10.1145/3231711

[21] Stephan Krusche, Nadine von Frankenberg, Lara Marie Reimer, and Bernd Bruegge. 2020. An Interactive Learning Method to Engage Students in Modeling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* (Seoul, South Korea) *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 12–22. https://doi.org/10.1145/3377814.3381701

[22] Timotej Lazar, Martin Možina, and Ivan Bratko. 2017. Automatic extraction of AST patterns for debugging student programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10331 LNAI (2017), 162–174. https://doi.org/10.1007/978-3-319-61425-0_14/TABLES/1

[23] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. A Context-Based Automated Approach for Method Name Consistency Checking and Suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, New York, NY, USA, 574–586. https://doi.org/10.1109/ICSE43902.2021.00060

[24] Artyom Lobanov, Timofey Bryksin, and Alexey Shpilman. 2019. Automatic classification of error types in solutions to programming assignments at online learning platform. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11626 LNAI (2019), 174–178. https://doi.org/10.1007/978-3-030-23207-8_33/FIGURES/2

[25] Thomas J. Mccabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2 (1976), 308–320. Issue 4. https://doi.org/10.1109/TSE.1976.233837

[26] Eerik Muuli, Kaspar Papli, Eno T onisson, Marina Lepp, Tauno Palts, et al. 2017. Automatic assessment of programming assignments using image recognition. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10474 LNCS (2017), 153–163. https://doi.org/10.1007/978-3-319-66610-5_12/TABLES/1

[27] Ramez Nabil, Nour Eldeen Mohamed, Ahmed Mahdy, Khaled Nader, Shereen Essam, et al. 2021. EvalSeer: An Intelligent Gamified System for Programming Assignments Assessment. In *2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*. IEEE, New York, NY, USA, 235–242. https://doi.org/10.1109/MIUCC52538.2021.9447629

[28] José Carlos Paiva, Paulo Leal, Álvaro Figueira, and Álvaro 2022 Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education (TOCE)* 22 (6 2022), 1–40. Issue 3. https://doi.org/10.1145/3513140

[29] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, et al. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 1093–1102.

[30] Dadi Ramesh and Suresh Kumar Sanampudi. 2022. An automated essay scoring systems: a systematic literature review. *Artificial Intelligence Review* 55 (3 2022), 2495–2527. Issue 3. https://doi.org/10.1007/S10462-021-10068-2/TABLES/9

[31] Stuart J Russell. 2010. *Artificial intelligence a modern approach*. Pearson Education, Inc., NJ, USA. 694–695 pages.

[32] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and Jose Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings* 2018-March (4 2018), 311–322. https://doi.org/10.1109/SANER.2018.8330219

[33] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. 2016. Question independent grading using machine learning: The case of computer program grading. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 13-17-August-2016 (8 2016), 263–272. https://doi.org/10.1145/2939672.2939696

[34] Fábio Rezende De Souza, Francisco De Assis Zampirolli, and Guiou Kobayashi. 2019. Convolutional neural network applied to code assignment grading. *CSEDU 2019 - Proceedings of the 11th International Conference on Computer Supported Education* 1 (2019), 62–69. https://doi.org/10.5220/0007711000620069

[35] Shashank Srikant and Varun Aggarwal. 2014. 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* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 1887–1896. https://doi.org/10.1145/2623330.2623377

[36] Arjun Verma, Prateksha Udhayanan, Rahul Murali Shankar, Nikhila KN, and Sujit Kumar Chakrabarti. 2021. Source-Code Similarity Measurement: Syntax Tree Fingerprinting for Automated Evaluation. In *The First International Conference on AI-ML-Systems* (Bangalore, India) *(AIMLSystems 2021)*. Association for Computing Machinery, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/3486001.3486228

[37] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, New York, NY, USA, 562–573. https://doi.org/10.1109/ICSE43902.2021.00059

[38] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. Association for Computing Machinery, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[39] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. 2019. Zero Shot Learning for Code Education: Rubric Sampling with Deep Learning Inference. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (7 2019), 782–790. Issue 01. https://doi.org/10.1609/AAAI.V33I01.3301782

[40] Xiaoming Zhai, Yue Yin, James W. Pellegrino, Kevin C. Haudek, and Lehong Shi. 2020. Applying machine learning in science assessment: a systematic review. *Studies in Science Education* 56, 1 (2020), 111–151. https://doi.org/10.1080/03057267.2020.1735757