Speeding Up Automated Assessment of Programming Exercises

Sami Sarsa
sami.sarsa@aalto.fi
Aalto University
Espoo, Finland

Juho Leinonen
juho.2.leinonen@aalto.fi
Aalto University
Espoo, Finland

Charles Koutcheme
charles.koutcheme@aalto.fi
Aalto University
Espoo, Finland

Arto Hellas
arto.hellas@aalto.fi
Aalto University
Espoo, Finland

ABSTRACT

Introductory programming courses around the world use automatic assessment. Automatic assessment for programming code is typically performed via unit tests which require computation time to execute, at times in significant amounts, leading to computation costs and delay in feedback to students. We present a step-based approach for speeding up automated assessment to address the issue, consisting of (1) a cache of past programming exercise submissions and their associated test results to avoid retesting equivalent new submissions; (2) static analysis to detect e.g. infinite loops (heuristically); (3) a machine learning model that evaluates programs without running them; (3) a machine learning model to evaluate programs without running them; and (4) a traditional set of unit tests. When a student submits code for an exercise, the code is evaluated sequentially through each step, providing feedback to the student at the earliest possible time, reducing the need to run tests. We evaluate the impact of the proposed approach using data collected from an introductory programming course and demonstrate a considerable reduction in the number of exercise submissions that require running the tests (up to 80% of exercises). Using the approach leads to faster feedback in a more sustainable way, and also provides opportunities for precise non-exercise specific feedback in steps (2) and (3).

CCS CONCEPTS

• **Social and professional topics** → *Computing education*; • **Applied computing** → *Interactive learning environments.*

KEYWORDS

automatic assessment, feedback, sustainability, automated assessment, source code, static analysis, machine learning, automated assessment, educational data mining

ACM Reference Format:
Sami Sarsa, Juho Leinonen, Charles Koutcheme, and Arto Hellas. 2022. Speeding Up Automated Assessment of Programming Exercises. In *The United Kingdom and Ireland Computing Education Research (UKICER) Conference (UKICER2022), September 1–2, 2022, Dublin, Ireland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3555009.3555013

1 INTRODUCTION

Traditionally, automatic assessment of student source code is done by running the students' code in isolated environments [1, 27, 42]; in these environments, the code is typically first built and then run against instructor-created unit tests to determine the correctness of the submission. One downside of this approach is that it can take a considerable amount of resources, especially in large programming courses such as MOOCs or when submissions contain infinite loops. Certainly, running the students code is not always necessary. Often simple features such as those based on static analysis could already tell whether something is wrong with the student's program.

In this work, we propose a novel automatic assessment approach that utilises a cache, static analysis, and machine learning to speed up automatic assessment. We propose that instead of running tests for every student submission, submitted source codes should first go over a series of checks to determine whether running tests is actually needed. When a student submits their program for evaluation, we first 1) check whether a cache already contains previously given feedback for equivalent code, 2) if not, we then analyse the student's code with static analysis methods to identify further issues with the code such as the existence of infinite loops[1], 3) if the program passes the static analysis checks, a machine learning model is used to predict whether the program is incorrect, and lastly 4) only if the program passes all the previous steps, we then run the code against a unit test suite.

We conduct a case study where we apply the approach to over 50,000 code submissions from a programming MOOC that uses the many small exercises approach [2]. Advantages of using our proposed approach over traditional automatic assessment that solely relies on unit tests are 1) faster and potentially more exact feedback to students, and 2) fewer resources needed which is both environmentally friendly and cheaper.

2 BACKGROUND

2.1 Automated assessment of programming exercises

Automated assessment systems have been used in programming education for decades [20, 25, 41], receiving plenty of attention

[1] Catching all cases of infinite loops is impossible due to the halting problem.

UKICER2022, September 1–2, 2022, Dublin, Ireland

ACM ISBN 978-1-4503-9742-1/22/09.
https://doi.org/10.1145/3555009.3555013

from both researchers and practitioners [1, 27, 42]. The earliest automated assessment systems existed during an era where remote access was not available and the systems required specific operators (e.g. [25]), while today's systems typically have an online interface [18, 32, 50] and often feature an online programming environment in which students can work on the exercises [43, 54].

Programming exercises are assessed with a handful of approaches, e.g. input-output testing and unit testing [1, 27], where the exercises and tests are crafted by course instructors or course material authors. Some automated assessment systems feature the possibility to assess student-written tests [18, 30], or to assess or limit the computational complexity of implemented algorithms [41, 50]. Unit testing can be performed in a variety of ways; in general, a secured environment is needed so that student code cannot influence the operating system or other programs [27]. To increase the speed of grading, if submitted code does not compile, systems may also omit running the tests altogether [15] and e.g. directly provide compilation errors as feedback instead.

2.2 Submission analytics

During the last decades, collection of source code and submission data has become more popular [29]. Analysis of such data has led into new insight e.g. on the types of errors that exist in students code [4, 38]. In general, these analyses often focus on static analysis of source code, which does not involve executing the analysed code [14]. Static analysis can be used to determine to what extent the source code has been written following common stylistic guidelines [16], to measure the complexity of the source code [28, 37, 40], and to identify common programming flaws [4, 5]. With certain restrictions, static analysis can also be used to detect infinite loops in source code [5, 11, 26, 36].

Submissions are also often studied for the purposes of finding similar submissions and to identify plagiarism [33, 37, 39, 45, 51], i.e. students copying solutions from each other or from external resources. Here, however, studies have highlighted limitations in plagiarism detection in that it might not be feasible especially in the context of smaller programming exercises [39, 48]. When considering how students may attempt to avoid plagiarism detection, there are a multitude of approaches, which include renaming variables and/or functions, reordering code, and adding meaningless code [17, 53].

Besides plagiarism detection, submission similarity and source code similarity have been used for providing feedback to students more efficiently [21, 44] and effectively [35]. While the more recent approaches that use machine learning on source code have relied on purpose-built program embeddings [3, 44], it is still unclear to what extent such embeddings generalise [34], and how source code should be represented (and modeled) seems to still seems to be a somewhat open question [24].

3 APPROACH

The proposed approach encompasses four different evaluation phases where student programs are checked for correctness. The steps are visualised in Figure 1. The steps are the following: (1) First, when a student submits an exercise for evaluation, a cache containing feedback given previously in the course for similar exercises is checked. If a match is found, feedback can be given instantly from the cache; (2) Second, if there are no direct matches in the cache, a static analysis of the student program is conducted to detect potential errors. Importantly, this does not require running the student code; (3) Third, if the student's program passes the previous checks, machine learning models that utilise features based on static analysis can be used to give feedback to students without running the program; and (4) if all of the previous checks pass, only then we actually run the program against instructor-created unit tests.

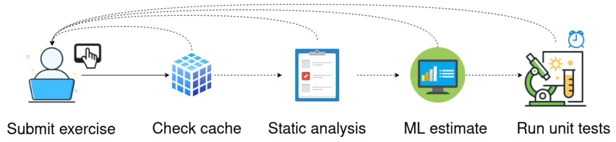

Figure 1: Our proposed evaluation sequence. Each step in the sequence is the final step if a match or problems are found in the step. 1. if the solution is in the cache, return cached test results; 2. if static analysis finds errors, return those; 3. if the ML-model is rather certain that there is a problem, return feedback from the model; 4. execute unit tests and return their results.

The first step (cache) is easy to implement using standard caching techniques such as LRU caching [12]. The similarity of previously submitted programs can be adjusted based on contextual needs: for example, one context might want to utilise string-based similarity metrics, while another might be happy with AST-based similarity.

The second step (static analysis) can encompass any type of static analysis, for example, identifying common syntactic errors and rule-based identification of infinite loops. Which static analysis approaches are utilised can depend on the programming language used and the needs of the specific context.

The third step (machine learning) relies on machine learning models that have been trained with static analysis based features using previously submitted student answers. What the models try to detect can again be adjusted based on the context: for example, one context might want to train a model to predict answer correctness, while another might be more keen to predict the presence of common misconceptions. A particular aspect of the third step is the possibility of the machine learning model giving an incorrect prediction, and thus students should be given an option to proceed to the fourth step if they disagree with the model's prediction. An important aspect here is that we are not predicting the correctness of learners' programs, but their incorrectness; to avoid fostering misconceptions, we do not want to risk providing feedback that falsely claims a program is correct when it is not.

The fourth final step (running tests) is the traditional test based automatic assessment approach and can further give "ground truth" about the correctness of the program. This is the most resource intensive step (considering only resources used for each individual submitted program) by a large margin.

The steps are intentionally described at an abstract level, allowing adaptation for the needs of specific contexts. As an example, checking for duplicates can be conducted using AST comparisons, which with some AST parsers require valid syntax – feedback on syntax errors could be provided at this point. At the same time, programming environments could also disallow submitting code that contain syntax errors, although this would not omit the need to do such checks also on the server.

Altogether, the goal is to speed up automated assessment. This provides benefits both by hastening the process of giving feedback (immediate error-correcting feedback when learning a task can lead to faster learning [22]) as well as saving computational resources and thus providing both economic and environmental benefits.

4 EVALUATION

4.1 Data

The dataset for our study comes from an open online introductory programming course organised by Aalto University. The workload of the course is 2 ECTS, which corresponds to roughly 50 to 60 hours of work. The course uses Dart as the programming language and features an online ebook with intertwined theory parts and programming exercises. There are 64 programming exercises in total. The programming exercises are worked on in an online programming environment embedded into the online ebook; the environment provides basic programming environment functionality such as syntax highlighting, autocompletion of code and executing code. The environment also allows working with standard input, provides functionality for submitting exercises for grading, and showing exercise grading feedback within the environment.

The course uses an automated assessment system to grade the submitted exercises and there are no upper limits for programming exercise submissions. The assessment of the programming exercises is based on a set of exercise-specific unit tests that verify that the functionality required in each specific exercise is present. Whenever grading a programming exercise, the automated assessment system creates a sandbox for security purposes. Grading a programming exercise typically takes between two to ten seconds; for programs with infinite loops or bugs leading to timeouts (e.g. waiting for standard input that is never given), we have used a timeout threshold of thirty seconds. The dataset contains 54,904 programming exercise submissions to a total of 64 exercises from a total of 725 learners. Of the submissions, 24,649 pass the tests, meaning that approximately 55% of the submissions are at least partially incorrect.

4.2 Research questions

We evaluate the proposed approach via a case study, answering the following research questions.

RQ1 How often can exercise solution results be retrieved from cache?
RQ2 How often can static analysis find errors in code submissions?
RQ3 How often can simple machine learning models reliably capture erroneous code submissions?

4.3 Approach

To answer RQ1, we explore two types of string-representations of the programming exercise submissions: (1) exact string matching, and (2) AST matching. The latter of the two requires valid syntax in order to form the AST, wherefore we include syntax checking already in this phase when presenting the results, even though it semantically falls under static analysis. In the present evaluation, we do not normalise variables names or do other code transformations.

To answer RQ2, we use static code analysis heuristics to find certain cases when the problem does not halt and to identify problems with semicolons. We keep the heuristics simple and merely check for existence of while true loops combined with non-existence of break or return clauses via regular expressions. In addition to looking at heuristics for non-halting programs, we look at a common error in many exercises in our data that is easily found with static analysis. The error consists of a semicolon directly after an “if”, “for” or “while” statement which causes the statement to not affect any forthcoming statements as demonstrated in Listing 1; this error is similar to the “Empty If Statements” discussed by Brown et al. [9]. Before computing the static analysis, we remove all the AST duplicates as these would bias the results and are removed in the cache step anyhow if present.

Listing 1: Typical extra semicolon error

```
if (x == "test"); {
  // do stuff falsely assuming x == "test"
}
```

To answer RQ3, we use a simple machine learning model to provide a fast-to-train baseline for the potential of catching incorrect solutions. Our model of choice is *Logistic regression* (LogReg), and as input for the model, we use two BoWs (Bag-of-Words) computed from submission ASTs. The two BoWs are token BoW and type label BoW, which are concatenated to form the final input.

We train a LogReg model (L-BFGS solver and 100 maximum iterations) per each exercise to identify erroneous submissions. Since we aim to capture erroneous submissions, we use incorrectness as the positive label and compute precision and recall to evaluate the goodness of the models. We choose precision and recall since we are interested in capturing as many erroneous examples as possible (maximising recall) while keeping the false positive rate as low as possible (maximising precision). Given that the label distribution is highly skewed and this affects both precision and recall, we also provide a naive *Majority Vote* model baseline results to account for the effect of data skew.

We train the models using separate training and test sets (80% training, 20% testing)[2]. As with static analysis, we remove the submissions caught in previous steps (duplicates and static analysis) to not bias the results and further, we only train and evaluate the models if the correctness skew is not extreme (minority label count in the test set amounts to at least 10% of the set size) in the resulting test set.

5 RESULTS

5.1 Cache catching and syntax errors

Out of the 54,904 programming exercise submissions, 19,374 (approximately 35%) were exact duplicates of other submissions. During parsing the submitted programming exercises into abstract

[2] We omit cross-validation since averaging over the exercises should balance out problems of having a single train-test split

syntax trees (ASTs), 1,686 submissions were identified to have syntax errors (approximately 3% of all submissions), and in total 31,434 (approximately 57%) of the ASTs were duplicates of other ASTs. When considering the individual exercises, we observed considerable variance in the quantity of exact duplicates and AST duplicates, where especially early on in the course there were more duplicates. Indeed, a traditional first programming exercise in the course, asking students to write a program that prints "Hello world!" had over 96% of both exact and AST duplicates.

Figure 2 shows the proportion of duplicates out of all submissions for each exercise, as both exact matches (blue solid line) and as AST matches (orange dashed line). Overall, depending on the exercise, the proportion of exact matches ranges from approximately 15% to nearly 97%. For the ASTs, the proportions are naturally somewhat larger, ranging from approximately 23% to nearly 97%.

The Figure 2 also shows the proportion of submissions with syntax errors for each exercise (green dotted line). Overall, there are considerably fewer syntax errors when compared to duplicates, ranging from nearly 0% to approximately 10% per exercise.

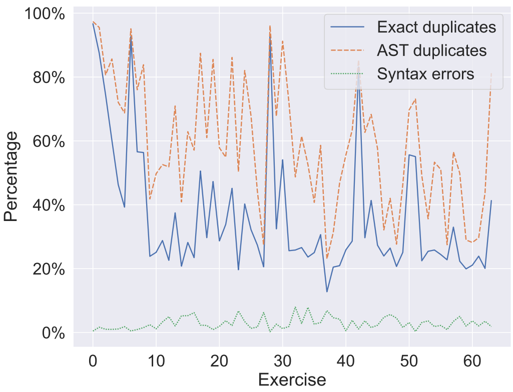

Figure 2: Proportion of duplicate programming exercise submissions and syntax errors for each programming exercise.

5.2 Static analysis

The static analysis conducted as a part of this work consisted of looking for semicolon problems (e.g. semicolon immediately after an "if" statement) and infinite loops (using a naive approach based on a regular expression). Out of the 21,784 unique programming exercise submissions after the removal of AST duplicates and submissions with syntax errors, 526 (approximately 2%) had semicolon issues, while 207 (approximately 1%) had infinite loop issues – some of these issues co-occured.

The proportions of these two types of issues are visualised in Figure 3 over the exercises. Overall, we observe that problems with semicolons are more common in the exercises than infinite loops found by our heuristics. In a few of the exercises, up to 18% of the submissions have semicolon problems while the largest proportion of infinite loops is approximately 5% for particular exercises.

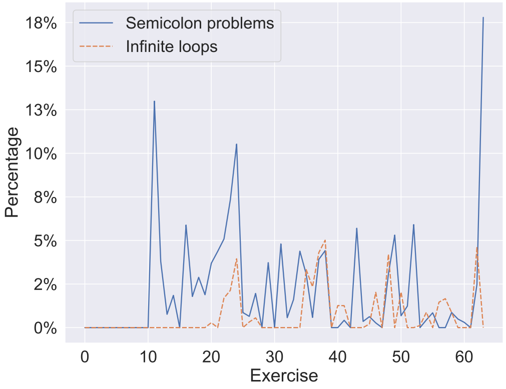

Figure 3: Static analysis issues identified in the submissions per exercise. Note that the y-axis values are shown up to 18%.

5.3 Machine learning based estimates of exercise correctness

When constructing machine-learning models for predicting exercise correctness, we used the 21,067 unique programming exercise submissions that remained after the removal of AST duplicates, syntax errors, as well as the submissions with found semicolon and infinite loop issues. Overall, when training the models, we observed a 0.95 PR-AUC (Area Under the Curve of a plot with Precision on y-axis and Recall on x-axis for all decision thresholds) for the LogReg model and 0.81 PR-AUC for the baseline majority vote model.

Figure 4 shows the averaged precision and recall thresholds over all the model scores for each programming exercise. Overall, the LogReg model achieves high precision and recall scores. As we are interested in capturing erroneous submissions while not allowing for many false positives, we present results with two different precision thresholds. At a threshold of 99% for precision, i.e. predicting a correct solution as an incorrect solution approximately once in one hundred submissions, the recall is approximately 10%. This means that approximately 10% of the incorrect solutions would be correctly identified as incorrect, thus needing to run unit tests for 90% of the incorrect submissions. At a threshold of 95% for precision we achieve approximately 70% recall overall for the exercises. This indicates that the LogReg model is able to capture around seven tenths of the erroneous non-duplicate exercise submissions (with 95% precision) without needing to run unit tests.

Note that we evaluated the models using only the data remaining in step 3 of the approach, i.e. having removed the duplicate programming exercise submissions as well as the submissions that were deemed to have issues during static analysis. We briefly explored the performance of the model using all 54,904 submissions, where we observed an improved performance for the LogReg model (0.97 LogReg PR-AUC) compared to the baseline majority vote model (0.73 MV PR-AUC).

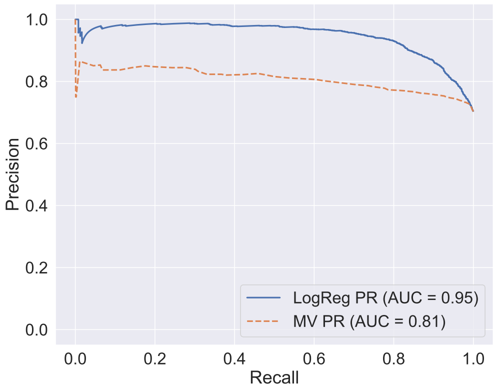

Figure 4: Logistic Regression error finding (incorrect solution is positive label) Precision-Recall Curve aggregated over each non-duplicate exercise where test sets consist of at least 10% minority labels. Majority Vote as a baseline allows seeing the effect of label skew in the curve.

6 DISCUSSION

6.1 Overview of results

Based on our analysis, the proposed approach considerably reduces the need to run tests when evaluating student submissions. Approximately one half of the submissions are equivalent to prior submissions when similarity detection is based on ASTs. The static analysis identifies a considerably smaller proportion of submissions with issues – in our data, less than 4% of the submissions (after removing duplicates) had issues that were identified using static analysis. The machine learning model shows promising results, with approximately half of the programs that have passed the previous two steps being correctly identified to fail the tests with a precision of 95% and recall of 70%. If a higher precision is desired, the recall will likely be lower as evidenced in Figure 4. The overall PR-AUC of the model was 0.95, which shows that the approach is feasible over different decision thresholds. The precision of 95% means that for twenty exercises the model predicts to be incorrect, one of them will actually be correct. The recall of 70% means that seven out of ten incorrect submissions that reach the machine learning step will be caught before sending them for testing.

Altogether, approximately 79% of the programming exercise submissions are caught by the approach before having to run the tests. Figure 5 illustrates these steps, showing the proportion of exercises at each step that will be correctly identified before the need to run the tests. If the course would be larger, the benefits would naturally be larger and vice versa.

6.2 Potential time shavings

In our implementations and evaluations with the present data, transforming a code to an AST took on average 7 milliseconds, checking whether a code was in the cache took on average 1 microsecond (starting with an empty cache), conducting the static analyses on a code took on average 49 microseconds, extracting features from an AST for machine learning purposes and receiving an ML estimate took on average 330 microseconds. All in all, despite additional overhead from the steps, the reductions in time are considerable, as approximately 79% of the code submissions could take less than 10 milliseconds to process. The computations were conducted on a single thread of a laptop CPU of one of the researchers.

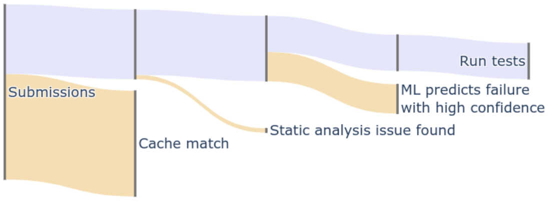

Figure 5: Sankey plot that shows the proportion of (incorrect) submissions captured at each step of the approach.

Considering that grading an exercise takes anywhere between two to thirty seconds in our traditional test based automatic assessment system, the approach would have saved between 24 and 361 hours of computing time and the precious time of the learners. This assumes that the learners would trust the machine learning step of the approach, where the learners would be given an option to disagree; if the learners would always want to run tests regardless of the machine learning results, the approach would still save between 18 and 265 hours.

6.3 Improvement directions

In the present study, we have presented the approach and evaluated it with a simple case study. We acknowledge that each step of the approach can be significantly tuned for better results when compared to our present study. As an example, for the cache matching, one could normalise variable names and partly the structure of the code [46]. Inspiration for this part could be also drawn from the research on plagiarism detection [33, 37, 39, 45, 47, 51] and on program equivalence checking [6, 49], although one would naturally want to avoid any approaches that rely on running the code.

For static analysis, one could include more sophisticated heuristics for infinite loop finding [5] and increase the number of checks for typical statically identifiable code issues, which has been done extensively before [4, 9, 19, 38]. One possible stream would be to also improve the feedback from compilation errors within the programming environment, which could potentially reduce the number erroneous submissions [7, 8]. We did, however, observe that submissions with syntax errors were relatively rare to begin with (approximately 3% of all submissions), which is most likely due to the code editor used in the course already indicating the presence of such errors while writing code and potentially also due to the many small exercises approach [2].

For the machine learning step, in our current evaluation, we have only trained a model that provides a binary prediction of correctness and which used relatively simple features. While the present evaluation shows the feasibility of the approach, we acknowledge that binary correctness feedback is far from ideal. There is, for example, no clear feedback on what to improve or what went wrong. This step could be considerably improved by training machine learning models that would use more advanced features for identifying mistakes, as discussed e.g. in [13, 23, 31, 52], and then provide feedback on the mistakes. We note, however, that some of these approaches require e.g. execution traces, which would require running the program and/or the tests and thus do not in their present state speed up assessment. It is also unclear to what extent the reported performance of these approaches has taken the existence of duplicates into account, which is an important preprocessing step in machine learning [10] – omitting this would likely inflate the model performance. As an example, in our case, we observed an increase in LogReg PR-AUC from 0.95 to 0.97 when including also the duplicate submissions; the difference compared to the baseline was more pronounced as the performance of the majority vote model decreased (from 0.81 to 0.73 PR-AUC).

Going beyond the improvements to the approach, we also see that parts of the approach could be distributed to learners' local environments. The static analysis and machine learning steps could be scaled and continuously run in the background of a programming environment to provide instant feedback similar to syntax checkers. This could go beyond education as feedback on potential problems may be of use in the professional context as well.

6.4 Limitations

There are some limitations to this work. Firstly, we evaluated the proposed approach with a context-specific case study using data from a single course. The course has many small exercises, which naturally leads to a higher number of similar submissions when compared to contexts that have larger programming projects.

Secondly, a part of the approach relies on a machine learning model that needs to be trained. Such training can take considerable amount of resources, depending on the used model. In the present evaluation, we used Logistic Regression, which took less than 30 seconds to train on a high-end laptop. However, had we utilised e.g. state of the art neural networks and larger datasets, it is possible that the training time would exceed the time saved by using the model. At the same time, smaller contexts do not necessarily have to train their own models. In the future, pre-trained models could be shared between contexts. It is possible, however, that training the model would still be meaningful as having a model can lead to faster feedback.

Thirdly, we acknowledge that there are a wide variety of ways for assessing programming exercises, including solely relying on static analysis [1]. Naturally, the proposed approach might not provide benefits in all cases. In the present evaluation, considerable benefits were observed, however.

Fourth, we acknowledge that with poor implementations, the amount of time saved could be less than in our present evaluations (or even negative). At the same time, we also note that there are multiple opportunities to further tune the approach by e.g. adjusting which steps to run for which exercises – based on our present implementations, we did not see this as relevant due to the already observed drastic differences.

7 CONCLUSION

In this work, we proposed an approach for speeding up automated assessment of programming exercises. We presented a case study of the approach where we used submission data from an open introductory programming course. We observed that the approach would reduce the number of submissions for which tests would be run by approximately 79%, leading to considerable reductions in the use of computational resources, and faster feedback to students. To summarise, our answers to the research questions are as follows.

RQ1. How often can exercise solution results be retrieved from cache? When transforming the submitted source codes to abstract syntax trees, we observed that approximately 57% of all of the submissions were duplicates. In our present study, we did not perform any transformations such as the normalisation approach proposed by Rivers and Koedinger [46], which would further improve the performance of the caching.

RQ2. How often can static analysis find errors in code submissions? After removing duplicate exercises identified with the caching approach, we further statically analysed the submissions to search for two types of errors. In the analysis, we observed that approximately 3.4% of the non-duplicate submissions had either infinite loops or semicolon issues. Again, we used relatively simple heuristics, and using a more fine-tuned approach would likely increase the quantity of identified errors.

RQ3. How often can simple machine learning models reliably capture erroneous code submissions? We built machine learning models per exercise to predict whether codes pass unit tests. We used relatively simple features and a relatively simple model, at least when compared to the state of the art. Even so, we obtained promising results, achieving an average of 0.95 PR-AUC score. When choosing 95% as the precision threshold, the recall score was 70%. The 95% precision means that on average for every twentieth prediction of submission being incorrect, the submission would actually be correct. The 70% recall means that on average seven out of ten submissions that are incorrect would not require assessment by unit tests as the machine learning model would catch these (assuming students trust the model).

As a part of our present and future work, we are taking the approach into use in our programming courses, and further evaluating with what types of programming courses the approach is most beneficial in – does it, for example, also work in a web development course with larger programming exercises? With the approach in use, we will be looking into evaluating the impact of faster feedback on students' learning in the context of introductory programming. Lastly, we are working on extending the machine learning models to identify specific common programming misconceptions to allow better and targeted formative feedback related to these misconceptions.

REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.

[2] Joe Michael Allen, Frank Vahid, Alex Edgcomb, Kelly Downey, and Kris Miller. 2019. An analysis of using many small programs in cs1. In *Proc. of the 50th ACM Technical Symposium on Computer Science Education*. 585–591.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proc. of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[4] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proc. of the 46th ACM technical symposium on computer science education*. 522–527.

[5] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008).

[6] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*. 13–24.

[7] Brett A Becker. 2016. An effective approach to enhancing compiler error messages. In *Proc. of the 47th ACM Technical Symposium on Computing Science Education*.

[8] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proc. of the working group reports on innovation and technology in computer science education*. 177–210.

[9] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A large scale repository of novice programmers' activity. In *Proc. of the 45th ACM technical symposium on Computer science education*. 223–228.

[10] Jason Brownlee. 2022. Data preparation for machine learning.

[11] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C Rinard. 2011. Detecting and escaping infinite loops with jolt. In *European Conf. on Object-Oriented Programming*. Springer, 609–633.

[12] Marek Chrobak and John Noga. 1999. LRU is better than FIFO. *Algorithmica* 23, 2 (1999), 180–185.

[13] Guillaume Cleuziou and Frédéric Flouvat. 2021. Learning Student Program Embeddings Using Abstract Execution Traces. *International Educational Data Mining Society* (2021).

[14] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. 2017. A comparison of static, dynamic, and hybrid analysis for malware detection. *J. of Computer Virology and Hacking Techniques* 13, 1 (2017).

[15] Karol Danutama and Inggriani Liem. 2013. Scalable autograder and LMS integration. *Procedia Technology* 11 (2013), 388–395.

[16] I.F. Darwin. 1988. *Checking C Programs with Lint.* O'Reilly & Associates.

[17] John L Donaldson, Ann-Marie Lancaster, and Paula H Sposato. 1981. A plagiarism detection system. In *Proc. of the twelfth SIGCSE technical symposium on Computer science education*. 21–25.

[18] Stephen H Edwards. 2003. Improving student performance by evaluating how well students test their own programs. *J. on Educational Resources in Computing (JERIC)* 3, 3 (2003).

[19] Stephen H Edwards, Nischel Kandru, and Mukund BM Rajagopal. 2017. Investigating static analysis errors in student Java programs. In *Proc. of the 2017 ACM Conf. on International Computing Education Research*. 65–73.

[20] George E Forsythe and Niklaus Wirth. 1965. Automatic grading programs. *Commun. ACM* 8, 5 (1965), 275–278.

[21] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.

[22] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of educational research* 77, 1 (2007), 81–112.

[23] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proc. of the Fourth (2017) ACM Conf. on Learning@ Scale*. 89–98.

[24] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.

[25] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3, 10 (1960), 528–529.

[26] Andreas Ibing and Alexandra Mai. 2015. A fixed-point algorithm for automated static detection of infinite loops. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. IEEE, 44–51.

[27] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proc. of the 10th Koli calling int. conf. on computing education research*. 86–93.

[28] Petri Ihantola and Andrew Petersen. 2019. Code Complexity in Introductory Programming Courses. In *Proc. of the 52nd Hawaii International Conf. on System Sciences, HICSS 2019*. 7662–7670.

[29] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, et al. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. *Proc. of the 2015 ITiCSE on Working Group Reports* (2015), 41–63.

[30] David Jackson and Michelle Usher. 1997. Grading student programs using ASSYST. In *Proc. of the twenty-eighth SIGCSE technical symposium on Computer science education*. 335–339.

[31] Sonja Johnson-Yu, Nicholas Bowman, Mehran Sahami, and Chris Piech. 2021. SimGrade: Using Code Similarity Measures for More Accurate Human Grading.. In *EDM*.

[32] Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The boss online submission and assessment system. *J. on Educ. Resources in Computing (JERIC)* 5, 3 (2005).

[33] Mike Joy and Michael Luck. 1999. Plagiarism in programming assignments. *IEEE Transactions on education* 42, 2 (1999), 129–133.

[34] Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. 2019. Assessing the Generalizability of Code2vec Token Embeddings. In *2019 34th IEEE/ACM International Conf. on Automated Software Engineering (ASE)*. 1–12.

[35] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2014. Strategy-based feedback in a programming tutor. In *Proc. of the computer science education research conf.* 43–54.

[36] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (1992), 323–337.

[37] Ronald J Leach. 1995. Using metrics to evaluate student programs. *ACM SIGCSE Bulletin* 27, 2 (1995), 41–43.

[38] David Liu and Andrew Petersen. 2019. Static analyses in python programming courses. In *Proc. of the 50th ACM Technical Symposium on Computer Science Education*. 666–671.

[39] Samuel Mann and Zelda Frew. 2006. Similarity and originality in code: plagiarism and normal variation in student assignments. In *Proc. of the 8th Australasian Conf. on Computing Education-Volume 52*. 143–150.

[40] Susan A Mengel and Vinay Yerramilli. 1999. A case study of the static analysis of the quality of novice student programs. In *Proc. of the thirtieth SIGCSE technical symposium on Computer science education*. 78–82.

[41] Peter Naur. 1964. Automatic grading of students' ALGOL programming. *BIT Numerical Mathematics* 4, 3 (1964), 177–188.

[42] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* 22, 3, Article 34 (jun 2022), 40 pages.

[43] Andrei Papancea, Jaime Spacco, and David Hovemeyer. 2013. An open platform for managing short programming exercises. In *Proc. of the ninth annual int. ACM conf. on International computing education research*. 47–52.

[44] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *International conf. on machine Learning*. PMLR, 1093–1102.

[45] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding Plagiarisms among a Set of Programs with JPlag. *J. of Universal Computer Science* 8, 11 (2002), 1016–1038.

[46] Kelly Rivers and Kenneth R Koedinger. 2013. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, Vol. 50.

[47] Francisco Rosales, Antonio García, Santiago Rodríguez, José L Pedraza, Rafael Méndez, and Manuel M Nieto. 2008. Detection of plagiarism in programming assignments. *IEEE Transactions on Education* 51, 2 (2008), 174–183.

[48] Simon, Oscar Karnalim, Judy Sheard, Ilir Dema, Amey Karkare, Juho Leinonen, Michael Liut, and Renée McCauley. 2020. Choosing code segments to exclude from code similarity detection. In *Proc. of the Working Group Reports on Innovation and Technology in Computer Science Education*. 1–19.

[49] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 3 (2012), 1–35.

[50] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students' learning using test my code. In *Proc. of the 18th ACM conf. on Innovation and technology in computer science education*. 117–122.

[51] Michael J Wise. 1992. Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing. *Acm Sigcse Bulletin* 24, 1 (1992), 268–271.

[52] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. 2019. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proc. of the AAAI Conf. on Artificial Intelligence*, Vol. 33. 782–790.

[53] Mengya Zheng, Xingyu Pan, and David Lillis. 2018. CodEX: Source Code Plagiarism Detection Based on Abstract Syntax Tree.. In *AICS*. 362–373.

[54] Daniel Zingaro, Yuliya Cherenkova, Olessia Karpova, and Andrew Petersen. 2013. Facilitating code-writing in PI classes. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 585–590.