Exploring Metrics for the Analysis of Code Submissions in an Introductory Data Science Course

Huy Anh Nguyen
hn1@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Michelle Lim
mlim1@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Steven Moore
stevenmo@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Eric Nyberg
ehn@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Majd Sakr
msakr@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

John Stamper
jstamper@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

ABSTRACT

While data science education has gained increased recognition in both academic institutions and industry, there has been a lack of research on automated coding assessment for novice students. Our work presents a first step in this direction, by leveraging the coding metrics from traditional software engineering (*Halstead Volume* and *Cyclomatic Complexity*) in combination with those that reflect a data science project's learning objectives (number of library calls and number of common library calls with the solution code). Through these metrics, we examined the code submissions of 97 students across two semesters of an introductory data science course. Our results indicated that the metrics can identify cases where students had overly complicated codes and would benefit from scaffolding feedback. The number of library calls, in particular, was also a significant predictor of changes in submission score and submission runtime, which highlights the distinctive nature of data science programming. We conclude with suggestions for extending our analyses towards more actionable intervention strategies, for example by tracking the fine-grained submission grading outputs throughout a student's submission history, to better model and support them in their data science learning process.

CCS CONCEPTS

• **Social and professional topics → Student assessment**; **Computing education**.

KEYWORDS

Coding Metrics, Linear Mixed Model, Data Science Education, Programming Analysis

ACM Reference Format:
Huy Anh Nguyen, Michelle Lim, Steven Moore, Eric Nyberg, Majd Sakr, and John Stamper. 2021. Exploring Metrics for the Analysis of Code Submissions in an Introductory Data Science Course. In *LAK21: 11th International Learning Analytics and Knowledge Conference (LAK21), April 12–16, 2021, Irvine, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3448139.3448209

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LAK21, April 12–16, 2021, Irvine, CA, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8935-8/21/04.
https://doi.org/10.1145/3448139.3448209

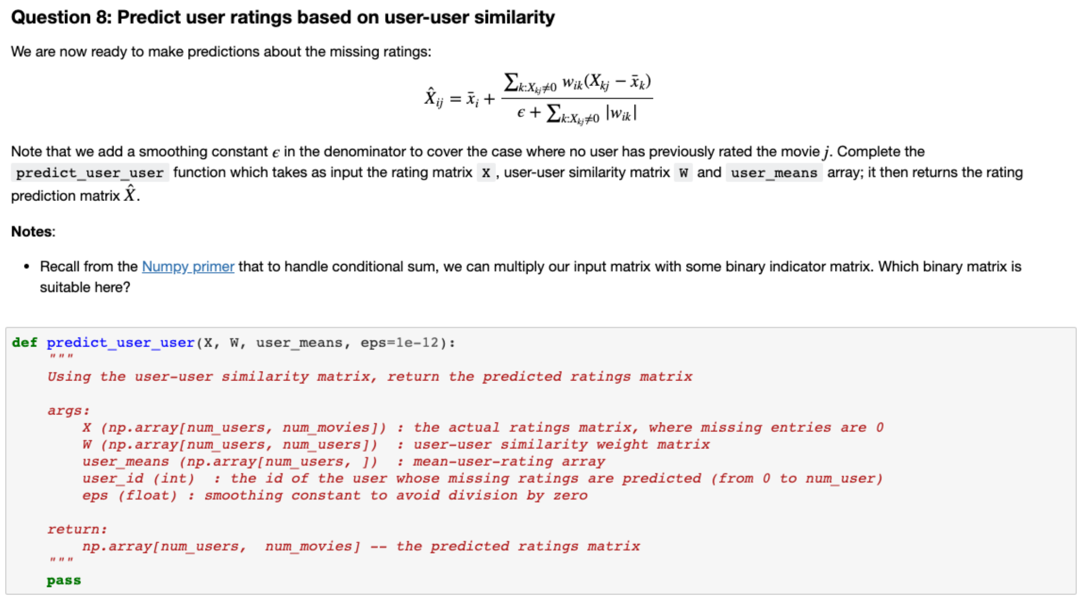
1 INTRODUCTION

Learning to program is not easy, and much research has gone into increasing success in teaching introductory computer science (CS1) [29]. Both solid conceptual and procedural knowledge are required to be good at programming; as a result, high failure and dropout rates are often reported in CS1 courses [21]. Learning to program in data science is even more difficult; unlike standard programming courses that majorly focus on core computer science topics (e.g., data structures and algorithms), data science is always tangled with mathematics, statistics, and specific aspects of various domains such as economics, linguistics, and climatology. Covering this complex skill set for students from an equally diverse set of backgrounds further adds to the challenge of data science instruction [4].

Advances in educational data mining and learning analytics may offer the key to resolve these challenges, as they have significantly improved the learner experience in CS1 courses over the years [8, 16, 28]. However, these techniques have not seen much adoption in data science education, where the focus of existing research remains at a high level of curriculum discussions [7, 32, 35] and case studies of successful course design [2, 31]. Towards promoting a more data-centered approach to evaluating and improving data science courses, in this work, we analyzed the code submissions of 97 students in an introductory data science project. Our goal is to identify suitable metrics that can accurately reflect the students' progress and identify areas of improvement for the project. Using a set of metrics from traditional software engineering, as well as those derived from the project's learning objectives, we examined the following research questions:

(1) How do the metric values vary across the project tasks, and how do they compare to the solution code metrics?
(2) How do the coding metrics relate to the submission runtimes?
(3) Which change in coding metric is indicative of a change in submission score?

Through answering these questions, this work contributes key insights into the large solution space that a data science task may have, as well as the importance of library calls in data science implementations. We also discuss the challenges in applying CS1



Question 8: Predict user ratings based on user-user similarity

We are now ready to make predictions about the missing ratings:

$$\hat{X}_{ij} = \bar{x}_i + \frac{\sum_{k:X_{kj}\neq 0} w_{ik}(X_{kj} - \bar{x}_k)}{\epsilon + \sum_{k:X_{kj}\neq 0} |w_{ik}|}$$

Note that we add a smoothing constant ϵ in the denominator to cover the case where no user has previously rated the movie j. Complete the `predict_user_user` function which takes as input the rating matrix `X`, user-user similarity matrix `W` and `user_means` array; it then returns the rating prediction matrix $\hat{X}$.

Notes:

- Recall from the Numpy primer that to handle conditional sum, we can multiply our input matrix with some binary indicator matrix. Which binary matrix is suitable here?

```
def predict_user_user(X, W, user_means, eps=1e-12):
    """
    Using the user-user similarity matrix, return the predicted ratings matrix

    args:
        X (np.array[num_users, num_movies]) : the actual ratings matrix, where missing entries are 0
        W (np.array[num_users, num_users])  : user-user similarity weight matrix
        user_means (np.array[num_users, ])  : mean-user-rating array
        user_id (int)   : the id of the user whose missing ratings are predicted (from 0 to num_user)
        eps (float) : smoothing constant to avoid division by zero

    return:
        np.array[num_users,  num_movies] -- the predicted ratings matrix
    """
    pass
```

Figure 1: An example project task with instructional text, docstring comments, and the template function for students to implement.

analytic techniques to this domain, and outline important next steps in better modeling and supporting students throughout their learning process.

2 BACKGROUND

2.1 Related Work

Automated grading techniques and systems have been an important part of recent advances in programming education [27, 36]. However, the output score alone may not be pedagogically meaningful to the instructors or the students. Towards more fine-grained student modeling through their code submissions, prior work has investigated the use of coding metrics as a means of quantifying the students' progress [18]. The explored metrics range from those in traditional software engineering (e.g., *Halstead Volume* [12], *Cyclomatic Complexity* [23]) to those more aligned with the assignment's goal, for example object-oriented metrics [6] in Java exercises. While many of these metrics only demonstrated weak correlations with the students' progress [15], they have seen usage in informing programming task designs and revisions. For instance, [10] proposed that each programming task can be measured in terms of *complexity*, expressed via the solution code metrics, and *difficulty*, captured by how students performed on it (e.g., failure rate, median solving time); based on these definitions, a significant discrepancy between the complexity and difficulty of a task would indicate that it is problematic and should be revised.

However, these prior works have all taken place in the context of introductory programming courses. It remains unclear if the traditional coding metrics are applicable in data science. For example, a study by [19] revealed that data science instructors and practitioners often focus on teaching the tabular data frame as the primary data structure, as opposed to those commonly seen in CS1 curriculum, such as linked lists and binary trees. The instructors also indicated that even the use of loops was not necessary, as canonical operations on data frames are supported by a wide range of vectorized library calls. These insights imply that data science codes would look very different from the programming code in typical CS1 courses. Characterizing this difference is among the primary goals of our research.

2.2 Course Description and Data Collection

Our analysis involves data collected from two semesters, Summer 2020 and Fall 2020, of a graduate-level introductory data science course at an R1 university in the northeastern United States. The course materials are divided into the conceptual components and the hands-on projects. Students learned from six conceptual units hosted on an e-learning platform, where each unit consists of reading assignments, practice activities and weekly quizzes. They also completed five individual Python coding projects in Summer 2020, which cover the following topics: (1) problem representation, (2) domain analysis and exploration, (3) domain data preparation, (4) machine learning and model performance, and (5) model deployment and comparison; in Fall 2020, a sixth project on evaluation optimization was added. Each project spans two weeks and involves the students implementing between 10-15 data science tasks on a template Jupyter notebook [17]. At any time before the project due date, students could submit their code to the autograding system and receive results after a few minutes, which include the grade for each task, the stack trace in case of an exception, and the first point of mismatch if the student's output differs from the reference output. There is no limit on the number of submissions, but there is a limit on the code runtime; if a student's submission exceeded the project's runtime threshold, it would not receive any points. In addition, each project imposes a constraint on the external libraries and packages that students can use.

In the scope of this work, we focused on submission data from Project 1. The project is an introduction to Numpy [14], Scipy [37] and Pandas [24], three popular data science libraries for data processing and numerical computations. In this project, students were given a dataset of 100,000 movie ratings and needed to use Pandas to perform basic dataframe manipulation (tasks 1-6), as well as Numpy/Scipy to implement the collaborative filtering recommendation algorithm (tasks 7-12). Figure 1 shows an example task in the template Jupyter notebook provided to students.

Over the two semesters, there were 97 graduate students who enrolled in one of six different STEM masters or doctoral programs. The students completed 1107 submissions to Project 1. 14 submissions had a compile error and received a score of 0; we also removed these submissions from our analyses as their code content could not be parsed. Therefore, our final sample consists of 1093 submissions from 97 students. There were 113 submissions with a full score of 100/100 coming from 76 students (multiple full-score submissions could belong to one student if they continued submitting after getting full scores to further optimize their code runtime).

Table 1: The list of metrics used in our analysis and their definitions.

Metric	Definition and Meaning
Halstead Volume	$V = (N_1 + N_2) \cdot \log(n_1 + n_2)$ where N_1, N_2, n_1, n_2 are the total number of operators, operands, distinct operators, and distinct operands respectively [12]. This metric represents the size, in bits, of space necessary for storing the program.
Cyclomatic Complexity	The number of linearly independent paths through a program's source code [23]. Our analysis uses the Radon library [1], which computes this metric as one plus the count of the following constructs: `if`, `elif`, `for`, `while`, `except`, `with`, `assert`, comprehension and boolean operator.
Logical Line of Codes	The number of executable statements in a program, measuring its size and development efforts [26].
AST Node Count	The number of nodes in the abstract syntax tree representation of a program, which reflects the count of distinct constructs in the code.
Library Call	The number of calls to functions and methods from external Python libraries (in Project 1, these are Numpy, Scipy and Pandas), which indicate the student's fluency with the provided libraries.
Solution Library Call	The number of Library Calls that are in common with those used in the solution code for the project (provided by the course instructors).

3 METHODS

To see how students' progress through the project can be reflected in their code submissions, we experimented with a set of six coding metrics as outlined in Table 1. Here we note that the first four metrics come from traditional software engineering measures and have also been used to assess students' works in introductory programming courses [9, 15]. The last two metrics are our custom metrics, motivated by Project 1's learning objective of familiarizing students with the common data processing libraries, which is an important skill for data science practitioners [19]. To compute them, we identified all the functions and method calls in the abstract syntax tree built from the input program, then traced their origins via the program's import statements. For example, if a program contains a call to `sp.diags` and the statement `import scipy.sparse as sp`, we can determine that `sp.diags` is a Scipy function call, which would count towards the *Library Call* value. In each student submission, we then recorded these metrics for each individual task as well as for the entire code content.

4 RESULTS

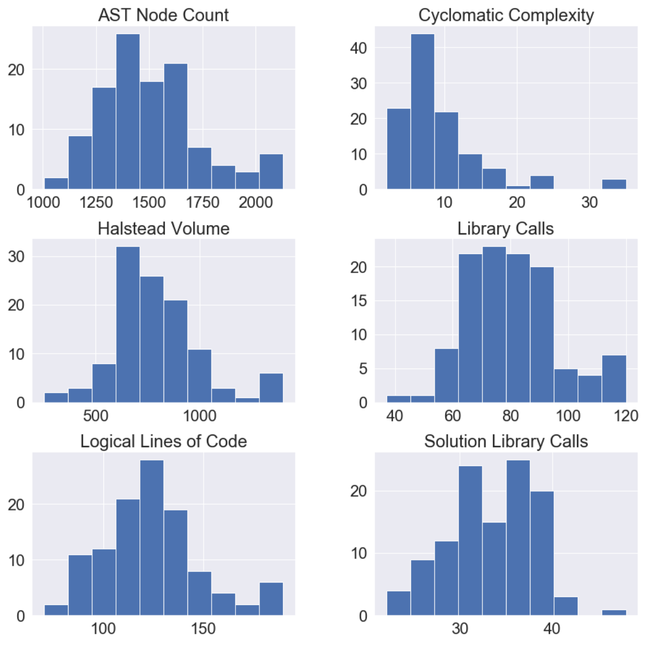


Figure 2: Histogram of each metric value distribution in the 113 full-score submissions. The x-axis denotes the metric values, and the y-axis denotes the number of students.

We first performed exploratory data analysis to visualize the distribution of each metric. As the submission codes naturally get longer and more complex when they include the implementations for more tasks, in this step, we only considered the 113 full-score submissions. In particular, we wanted to see how the full code contents in these submissions vary in terms of the chosen metrics. Figure 2 shows that there is indeed a wide variability across all metrics, where the maximum metric value is always at least twice as large as the minimum. This variability indicates that there are notable differences among the submissions, even though they were all at the completion stage of the project. For example, based on the *Logical Lines of Code* distribution, we observed that some students only wrote 80 lines to implement all 12 tasks, while others took more than 160 lines. To understand what these differences imply in terms

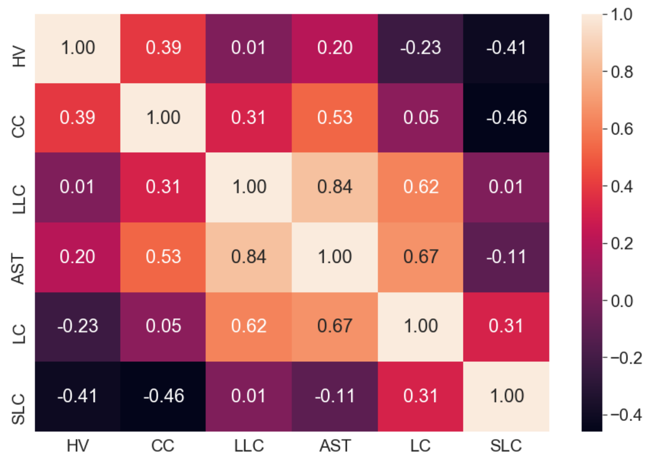


Figure 3: Spearman correlation matrix for the 6 metrics, based on the 113 full-score submissions. HV, CC, LLC, AST, LC and SLC indicate *Halstead Volume, Cyclomatic Complexity, Logical Lines of Code, AST Node Count, Library Call* and *Solution Library Call* respectively.

of student evaluation, our next step is to build regression models from the chosen metrics. However, we first wanted to identify any internal correlations among the metrics themselves. If a group of metrics had a high degree of correlation, they could be considered as conveying roughly the same information, and we would only need to select one metric from that group for subsequent analyses.

Figure 3 shows the pairwise correlation heatmap, where the following three pairs of metrics have a high correlation (with $r \geq 0.60$): *Library Call - Logical Lines of Code* ($r = 0.65$), *AST Node Count - Library Call* ($r = 0.75$), and *AST Node Count - Logical Lines of Code* ($r = 0.84$). In other words, *Library Call*, *AST Node Count*, *Logical Lines of Code* are pairwise strongly correlated. Therefore, we decided to select *Halstead Volume*, *Cyclomatic Complexity*, *Library Call* and *Solution Library Call*, which are not strongly correlated with one another, as our final set of metrics.

4.1 RQ1: How do the metric values vary across the project tasks, and how do they compare to the solution code metrics?

In this analysis, we only considered the 113 full-score submissions, as they are functionally equivalent to the solution, i.e., they had correct implementations for all the tasks; submissions with lower scores either did not implement some tasks or implemented them incorrectly, making their code metrics difficult to interpret. In each full-score submission, we computed the coding metrics for each individual task implementation. We then plotted the distribution of these metric values per task in Figure 4, where each point on the line graph indicates the mean value for a given task. The shaded region denotes the 95% confidence interval of the metric distribution. We also indicated the solution code metrics in red dots to facilitate the comparison with the students' codes.

For the first three metrics, while the line plots each follow a different pattern, they all have peaks at task 6 and 11, which imply

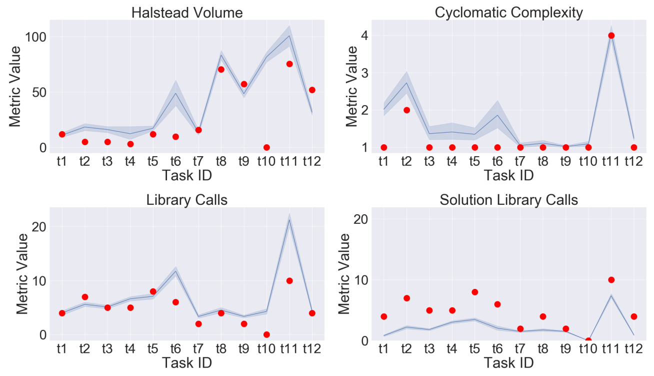


Figure 4: Distribution of metric values across tasks from the full-score submissions (denoted by the line plot) and the solution code (denoted by the red dots).

that these were, to students, the more complex tasks in the project. The solution code metric for task 11 is also high, so this task was indeed intended to be challenging. On the other hand, there is no notable peak at task 6 for the solution code, indicating that most students' implementations were more complex than necessary. A similar gap between the students' codes and solution code is observed in task 10. This task can be solved by simply applying the function from task 9 to the transpose of the input matrix; therefore, the solution only contains one line of code. However, most students did not realize this connection and instead implemented task 10 from scratch, resulting in more complicated code. Finally, the line plot for *Solution Library Call* shows that, in most tasks, only about 30-50% of the library calls used in the solution code were in common with those used in the students' codes. In other words, students were able to solve the project tasks with many library calls not used by the solution.

In summary, we have identified variations in the coding metrics across tasks, where task 11 could be considered the most complex. At the same time, there were notable gaps in the code metrics between the students' and the solution's implementations of task 6 and 9. To better understand these metric differences, we next examined their relationships with the outcome of each submission, which can be measured by the runtime and the output score.

4.2 RQ2: How do the coding metrics relate to the submission runtimes?

For the same rationale outlined in RQ1, we only considered the 113 full-score submissions in this analysis. We then constructed a linear regression model where the independent variables are the four metric values of each submission, and the dependent variable is the submission runtime. Our results in Table 2 showed that *Library Call* is a positive and significant predictor of the code runtime. However, we also noted that the R^2 value of this regression model is quite low (0.102), so our current coding metrics alone were not able to fully capture the variance in submission runtime.

Table 2: Results of the linear regression that predicts the submission runtimes based on the submission coding metrics. (*) and (**) indicate significance at the 0.05 and 0.01 levels respectively.

	Coef	Std Err	*t*	95% CI
Intercept	107.1752	40.573	2.642 (*)	(26.717, 187.633)
Halstead Volume	-0.0114	0.022	-0.528	(-0.054, 0.031)
Cyclomatic Complexity	0.6093	0.815	0.747	(-1.007, 2.226)
Library Call	0.7443	0.254	2.934 (**)	(0.241, 1.247)
Solution Library Call	-0.9935	0.959	-1.036	(-2.894, 0.907)

Table 3: Results of the linear mixed model that predicts the change in submission scores based on the changes in metric values. (*) and (**) indicate significance at the 0.05 and 0.01 levels respectively.

	Coef	Std Err	z	95% CI
Intercept	12.788	0.516	24.800 (**)	(11.777. 13.798)
Halstead Volume	0.013	0.008	1.672	(-0.002, 0.029)
Cyclomatic Complexity	-0.618	0.445	-1.389	(-1.489, 0.254)
Library Call	0.414	0.175	2.364 (*)	(0.071, 0.757)
Solution Library Call	-0.072	0.418	-0.172	(-0.891, 0.747)
Group Var	46.210			

4.3 RQ3: Which change in coding metric is indicative of a change in submission score?

For each student, we considered all of the unique score thresholds that they obtained in the course of their submissions. For example, if student A made four submissions s_1, s_2, s_3, s_4 with the following scores: 10, 10, 50, and 100, A would have three score thresholds - 10, 50, and 100. Next, we recorded the changes in score and in coding metric values between the submission at each threshold. With the example of A, we would record how much each coding metric changed when their score went from 10 to 50, and when their score went from 50 to 100. If there are multiple submissions at a given threshold (e.g., s_1 and s_2 both scored 10 points), we would consider only the first submission at that threshold (i.e., s_1), in order to fully capture the range of code changes that the student made to advance to the next score threshold.

After extracting these score and coding metric differences, we set up a linear mixed model as follows:

$$\Delta S \sim \Delta HV + \Delta CC + \Delta LC + \Delta SLC + (1 \mid SID), \qquad (1)$$

where $\Delta S, \Delta HV, \Delta CC, \Delta LC$ and ΔSLC are the differences between each submission and the submission at the previous score threshold (from the same student) in terms of score, *Halstead Volume*, *Cyclomatic Complexity*, *Library Call* and *Solution Library Call* respectively. Because each student contributed multiple data points, we used the student identifier *SID* as the random effect. This factor can be considered as a representation of each individual student's coding style. For example, one student may chain multiple function calls in one line, while another saves each call output to a variable; these different styles would lead to different baseline metric values, which the random effect can account for.

With the submission selection criteria outlined for the hypothetical student A earlier, we collected 365 data points for the linear model (1) and implemented it using Python's `statsmodel` library [33]. Our results in Table 3 showed that LC, the change in *Library Call*, is a significant positive predictor of the change in score. In other words, an increase in the number of library calls used in the student's submission is most indicative of an increase in score.

5 DISCUSSION AND CONCLUSION

Our work investigated the coding metrics of student submissions in an introductory data science project, as a first step in connecting data science education and programming analysis. Through examining a combination of traditional software engineering metrics (*Halstead Volume*, *Cyclomatic Complexity*) and data science-specific metrics (*Library Call*, *Solution Library Call*), we have identified the metrics that are indicative of the task complexity, submission runtime and submission score. Here we further discuss how these findings highlight the distinctive features of data science programming and how they can contribute to providing instructional feedback in an introductory course.

First, we observed a wide variety in coding metric values even among the full-score submissions, which all perform the same tasks. This diversity highlights a key difference between data science and introductory programming: that the solution space of a data science task can be large, even if the task is well-defined with a fixed correct output (such as those used in this project). This insight is further supported by comparing the students' code metrics with the solution's; our visualization in Figure 4 showed that, in most cases, the students' code contents were distinct from the solution codes. Most notably, the distribution of *Solution Library Call* remains low throughout the tasks, indicating that, even when Project 1 was intended to teach students about using data science libraries, the library calls that they chose for their implementations were largely different from what the instructors had anticipated. While prior research in introductory programming has typically used the *distance* between a student's code and the solution code to measure their progress [30], our findings indicate that its effectiveness in data science may be limited. Students may still perform well even when they they took different approaches from the solution. It is possible that deviation from the solution can still be a meaningful metric, but should be measured at a higher level than the code contents. For example, in introductory programming, [20] has proposed a mapping from the abstract syntax tree nodes to their corresponding high-level concepts. Future works could construct a similar mapping for data science codes, in order to capture the conceptual gap between a student's code and the solution's, which may better reflect their progress.

At the same time, identifying notable gaps between the students' code metrics and the solution's can also guide course improvement strategies. If most students' codes had lower metric values than the solution code for a particular task, the students likely made use of a shortcut through the task which the instructors did not anticipate. In this case, the instructors should either redesign the task to prevent this shortcut, or adjust its score weighting to account for the lower-than-expected difficulty. While learning curve analysis has also been applied to detect students' shortcut approaches [13], this technique requires fine-grained interaction data and a knowledge component model [25, 34]; our visualization method can be considered a step in a similar direction, but relies only on the student's submission codes. On the other hand, the opposite scenario may occur where the solution code metrics were lower than those of most students, which we observed in task 6 and 10. The solution for task 10, in particular, was very simple, but many students followed a more complicated pathway because they did not realize the connection with previous tasks. This is a potential area for providing adaptive hint and feedback, where the autograder could display a message such as "Your code is more complex than necessary. Think about how you can utilize your implementation of task 9." As prior works have demonstrated the effectiveness of code hints [30] (especially with textual explanations [22]) in programming exercises, we expect a similar effect of the proposed hints in our domain of data science education.

In our next analysis, we ran a linear regression model to examine how each coding metric contributed to the submission runtime. We noted that the model's R^2 is quite low, likely because the runtime duration depends on the input dataset as well as the number of test cases, which our model did not consider. On the other hand, we still identified *Library Call* as a positive and significant predictor of the submission runtime. This effect can be explained by the vectorized nature of the library calls from Numpy/Scipy and Pandas [14]; in particular, these library calls were designed to operate on an entire vector or matrix. Despite being highly optimized, multiple library calls still involved multiple traversals over the input dataset (with 100,000 rows in this project), which would lead to higher runtime.

Another measure of submission outcome that we investigated is the output score; in particular, we wanted to see which change in coding metric closely aligned with the change in score, when the student's score did change. To this end, we set up a linear mixed model that predicted the change in score based on the change in each coding metric, with the student ID acting as the random effect. We found that *Library Call* is again a positive significant predictor. In other words, the most indicative factor of an increase in student score is that they used more library calls. A possible explanation is that each project task is independent and has its own implementation, so a student is more likely to use additional library calls when they work on a new task, rather than when they debug an already implemented task – in this case, they are also likely to get higher score from the new task. More generally, this finding highlights a similar observation from [19] – in their study of how data science practitioners taught the subject, the authors noted an emphasis on "connecting existing APIs together in order to shape them for the analytic tasks at hand," as opposed to implementing new functions or classes, which is more common in traditional programming. In other words, an important learning objective for data science novices is to acquire fluency in common data science APIs; this is another distinctive feature of data science that should be considered for research on coding analysis in this domain.

As a first step towards exploring the value of coding metrics in data science education, our work has certain limitations that we plan to address in future studies. First, to create a meaningful context for metric comparison, the analysis so far has focused only on a subset of the 1093 submissions to Project 1. For the next step, we would like to investigate the full submission history of each student in order to identify those who need additional support in the course. This task would involve examining more fine-grained submission outputs, such as the failed test cases or error message logs [3], and potentially integrating other student behavior data in the course that can reflect their learning habit [11]. Second, we plan to collect data from other data science courses at different institutions to validate the generalizability of our findings. Third, we should emphasize that data science is a highly interdisciplinary area, where the nature of the code in each sub-domain may be very distinct. For example, a project on web scraping would place less emphasis on vectorized operations but more on text parsing and cleaning. Therefore, it is important to examine each sub-domain separately, while also identifying the core "data science thinking" components [5] that transfer between sub-domains.

As demand for data science rapidly increases, universities and organizations are actively expanding their DS course offerings. However, our research indicates that a more evidence-based approach to teaching and learning data science is needed, due to the distinctive nature of this domain. In particular, we found that the number of library calls, rather than traditional metrics used in CS1 courses, is a significant predictor of code runtime and changes in score. This result reflects the importance of utilizing existing data science libraries in solving analytic problems; it can thus be used by instructors and department heads to revise their curriculum and assessment mechanisms accordingly. We also showed that comparing the metric values of the student codes and the reference code can reveal which tasks are easier or harder than expected, which further aids instructional revision. These results in turn lay the foundation for follow-up work on implementing data-driven and timely interventions, which would ultimately contribute to a scalable workflow for personalized student support in data science education.

REFERENCES

[1] [n.d.]. Radon. https://github.com/rubik/radon.
[2] Craig Anslow, John Brosz, Frank Maurer, and Mike Boyes. 2016. Datathons: an experience report of data hackathons for data science education. In *Proceedings*

of the 47th ACM Technical Symposium on Computing Science Education. 615–620.

[3] Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. 2002. An experience in integrating automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin* 34, 4 (2002), 125–128.

[4] Robert J Brunner and Edward J Kim. 2016. Teaching data science. *Procedia Computer Science* 80 (2016), 1947–1956.

[5] Longbing Cao. 2018. Data Science Thinking. In *Data Science Thinking*. Springer, 59–90.

[6] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[7] Richard D De Veaux, Mahesh Agarwal, Maia Averett, Benjamin S Baumer, Andrew Bray, Thomas C Bressoud, Lance Bryant, Lei Z Cheng, Amanda Francis, Robert Gould, et al. 2017. Curriculum guidelines for undergraduate programs in data science. *Annual Review of Statistics and Its Application* 4 (2017), 15–30.

[8] Nicholas Diana, Michael Eagle, John Stamper, Shuchi Grover, Marie Bienkowski, and Satabdi Basu. 2017. An instructor dashboard for real-time analytics in interactive programming assignments. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*. 272–279.

[9] Tomáš Effenberger, Jaroslav Cechák, and Radek Pelánek. 2019. Difficulty and Complexity of Introductory Programming Problems. (2019).

[10] Tomáš Effenberger, Jaroslav Čechák, and Radek Pelánek. 2019. Measuring Difficulty of Introductory Programming Tasks. In *Proceedings of the Sixth (2019) ACM Conference on Learning@ Scale*. 1–4.

[11] Seth Copen Goldstein, Hongyi Zhang, Majd Sakr, Haokang An, and Cameron Dashti. 2019. Understanding how work habits influence student performance. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 154–160.

[12] Maurice Howard Halstead et al. 1977. *Elements of software science*. Vol. 7. Elsevier New York.

[13] Erik Harpstead and Vincent Aleven. 2015. Using empirical learning curve analysis to inform design in an educational game. In *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*. 197–207.

[14] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[15] Petri Ihantola and Andrew Petersen. 2019. Code complexity in introductory programming courses. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*.

[16] 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. In *Proceedings of the 2015 ITiCSE on Working Group Reports*. 41–63.

[17] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows.. In *ELPUB*. 87–90.

[18] Pardha Koyya, Young Lee, and Jeong Yang. 2013. Feedback for programming assignments using software-metrics and reference code. *International Scholarly Research Notices* 2013 (2013).

[19] Sean Kross and Philip J Guo. 2019. Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–14.

[20] Andrew Luxton-Reilly and Andrew Petersen. 2017. The compound nature of novice programming assessments. In *Proceedings of the Nineteenth Australasian Computing Education Conference*. 26–35.

[21] Sohail Iqbal Malik. 2018. Improvements in introductory programming course: action research insights and outcomes. *Systemic Practice and Action Research* 31, 6 (2018), 637–656.

[22] Samiha Marwan, Joseph Jay Williams, and Thomas Price. 2019. An Evaluation of the Impact of Automated Programming Hints on Performance and Learning. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 61–70.

[23] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[24] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. Austin, TX, 51–56.

[25] Huy Nguyen, Yeyu Wang, John Stamper, and Bruce M McLaren. 2019. Using Knowledge Component Modeling to Increase Domain Understanding in a Digital Learning Game. *International Educational Data Mining Society* (2019).

[26] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC counting standard. In *Cocomo ii forum*, Vol. 2007. Citeseer, 1–16.

[27] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 92–97.

[28] Thomas Price, Baker Franke, Shuchi Grover, and Monica M McGill. 2020. Using Data to Inform Computing Education Research and Practice. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 175–176.

[29] Keith Quille and Susan Bergin. 2019. CS1: how will they do? How can we help? A decade of research and practice. *Computer Science Education* 29, 2-3 (2019), 254–282.

[30] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.

[31] Jeffrey Saltz and Robert Heckman. 2016. Big Data science education: A case study of a project-focused introductory course. *Themes in science and technology education* 8, 2 (2016), 85–94.

[32] Jeffrey S Saltz, Neil I Dewar, and Robert Heckman. 2018. Key concepts for a data science ethics curriculum. In *Proceedings of the 49th ACM technical symposium on computer science education*. 952–957.

[33] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*.

[34] John C Stamper and Kenneth R Koedinger. 2011. Human-machine student model discovery and improvement using DataShop. In *International Conference on Artificial Intelligence in Education*. Springer, 353–360.

[35] Rong Tang and Watinee Sae-Lim. 2016. Data science programs in US higher education: An exploratory content analysis of program description, curriculum structure, and course focus. *Education for Information* 32, 3 (2016), 269–290.

[36] Leo C Ureel II and Charles Wallace. 2019. Automated Critique of Early Programming Antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 738–744.

[37] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.