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:

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, 25, 26]. 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) What 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
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 came from traditional software engineering measures and have metrics as outlined in Table 1. Here we note that the first four metrics were defined in this study.

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

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halstead Volume</td>
<td>$V = (N_1 + N_2) \cdot \log (n_1 + n_2)$ 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.</td>
</tr>
<tr>
<td>Cyclomatic Complexity</td>
<td>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.</td>
</tr>
<tr>
<td>Logical Line of Codes</td>
<td>The number of executable statements in a program, measuring its size and development efforts [26].</td>
</tr>
<tr>
<td>AST Node Count</td>
<td>The number of nodes in the abstract syntax tree representation of a program, which reflects the count of distinct constructs in the code.</td>
</tr>
<tr>
<td>Library Call</td>
<td>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.</td>
</tr>
<tr>
<td>Solution Library Call</td>
<td>The number of Library Calls that are in common with those used in the solution code for the project (provided by the course instructors).</td>
</tr>
</tbody>
</table>

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

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
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 ($r \geq 0.60$): 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 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.
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:

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

(1)

where $\Delta S$, $\Delta HV$, $\Delta CC$, $\Delta LC$ and $\Delta SLC$ 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 choose 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
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 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**


