Title
Measuring Formative Learning Behaviors of Introductory Statistical Programming in R via Content Clustering

Author
Roberts, Shane

Publication Date
2015

Peer reviewed|Thesis/dissertation

University of California
Los Angeles

Measuring Formative Learning Behaviors of Introductory Statistical Programming in R via Content Clustering

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Statistics

by

Shane William Strouth Roberts

2015

Abstract of the Thesis

Measuring Formative Learning Behaviors of Introductory Statistical Programming in R via Content Clustering

by

Shane William Strouth Roberts

Master of Science in Statistics

University of California, Los Angeles, 2015

Professor Robert Gould, Chair

Understanding student learning is an open problem in the teaching of introductory statistical programming. Formative learning is observed when analyzing the interaction between the learning environment and the student. This can be operationalized by viewing how students respond to the error messages they receive while programming. Current California Common Core State Standards: Mathematics (CA CCSSM) highlight perseverance as an overarching habit of a productive mathematical thinker. Perseverance can be measured as a type of formative learning by measuring the time and attempts that students use to correct errors. The MOBILIZE project promotes statistical programming at the high school level in the Los Angeles Unified School District while following the CA CCSSM. Logs of high school student statistical programming during the 2013-2014 school year were collected along with the errors that occurred. Using these logs, error blocks were formed that follow a student's interaction with an error message. With the error blocks we were able to observe perseverance by students on multiple days of curriculum. Our findings suggest that there was an increase in perseverance, because students increased in time spent and attempts made to correct an error as the number of days of programming curriculum increased. Additionally, students who showed more perseverance were more likely to eventually fix an error. Descriptive variables were explored to provide background for the variation in student programming errors.

The thesis of Shane William Strouth Roberts is approved.

Frederic P. Schoenberg

Hongquan Xu

Robert Gould, Committee Chair

University of California, Los Angeles

2015

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Studies of programming education often report poor learning results for students in introductory programming courses. However, improvement of these introductory courses can be achieved with careful evaluation of a student's interaction with a new programming environment. Creation of learning metrics is a valuable tool to aid improvement, but the course curriculum and programming language of instruction must be carefully considered. There are numerous programming languages that are taught at the introductory level, and numerous purposes that are the focus of introductory programming courses. A clear consensus has not been reached on the optimal language or curriculum to approach introductory programming. Reviews of the current literature have suggested that a case by case evaluation is needed until a comprehensive study of the various options for introductory programming has been concluded. For this reason, the subject of improving and evaluating introductory programming education continues to be an open research problem[1, 2, 3, 4].

The programming language R "is an integrated suite of software facilities for data manipulation, calculation, and graphical display"[5]. While not strictly a statistical programming language, R is often used for statistical analysis and is a popular programming language for introductory statistical programming courses. Unfortunately, much of the current research on the evaluation of introductory programming is focused on general computer programming courses. While evaluating introductory statistical programming (such as that produced in R), differences between general and statistical programming courses need to be considered to properly measure learning.

Program results, code style, line count, compilation time, comment usage, and simplicity are just some of the many possibilities for the evaluation of student generated code in general introductory programming courses. A set of these metrics can be used

to deduce student learning in the course. Measurement for most of these metrics requires an environment that generates multiple lines of code as chunks. These metrics are possible at the introductory level because the environment used in introductory programming courses most often contains an explicit compilation step. An explicit compilation is when the translation of code into machine processing is completed with an explicit action rather than automatically after the completion of a line. The resulting code is usually written in chunks of multiple lines of code, and then compiled. There is not an explicit compilation step when programming R in the console for introductory statistical programming. Instead, introductory level programming in the R environment is a series of single lines of code compiled individually. This style of programming is referred to as scripting. The resulting code from scripting and explicit compilation environments differ in structure. New metrics are needed so that we can offer measurements of learning in statistical programming in R at an introductory level.

In this paper, data from the Mobilize Project is utilized to explore the topic of measuring learning in an introductory statistical programming course. The Mobilize project is a NSF funded project by the University of California, Los Angeles in partnership with the Los Angeles Unified School District. The Mobilize project includes an introductory statistical programming course - Exploring Computer Science(ECS). ECS - the Mobilize project created a six week module for ECS to teach computing with data. The goal of Mobilize is "to strengthen computer science instruction throughout the Los Angeles high school educational system and to develop innovative methods for educating and engaging students in computational thinking and data analysis"[7]. ECS is one part of this larger Mobilize project that was founded in 2010. ECS is a new high school level curriculum that is designed to pair computational thinking with relatable examples for high school students. Within the curriculum the six week module on statistical programming is offered using the programming language R. The current syllabus for the ECS curriculum, called Mobilize Prime, is based on a programming style of single line inputs of pre-written functions for conducting data analysis. This style is of the scripting type rather than an explicit compilation type. In 2013-2014, the Mobilize Prime Module was taught by three ECS teachers with two classes each. A total of 140 students were enrolled in the six classes. The students were ages 14-18, and were enrolled in the class based on councilor

recommendation. Demographics of the student are reasonably consistent with the school district at large. The ECS course is not designated as advanced nor remedial, so a range of student ability is expected to be found in the students. Logs of each students' programming during the statistical programming unit were collected and will be used for analysis in this paper.

We will propose a method for the formative assessment of learning in introductory statistical programming of R. Formative assessment differs from the usual method of measuring learning which is summative assessment. Summative assessment is the measurement of learning that occurs at the end of a timeline, and is not part of the learning process. Examples of summative assessment are standardized tests, and final projects. Instead, formative assessment occurs during the learning process and is often paired with automatic feedback to improve student learning. In this paper, we will examine the formative learning that surrounds the error messages students receive while programming in R. We will take advantage of the scripting style of programming to view how students behave when confronted with the automatic error messages. We will focus on how long students spend on correcting errors, how many attempts they need to correct an error, and the proportion of errors that are successfully corrected. The student interaction with error messages and the process of attempting to correct errors will be refered to as **perseverance**. Where students who spend more time on correcting errors, and attempt more corrections, are showing more perseverance in their learning. Perseverance is listed as the first "Overarching habit of mind of a productive mathematical thinker" in the Current California Common Core State Standards: Mathematics (CA CCSSM)[8]. For this reason, our analysis will be exploring the realization of an important standard for evaluating the curriculum. In order to view these metrics, we clustered lines of code based on content into **error blocks**. Each error block will contain an initial error, repeated errors, and corrected error if the student was able to successfully correct the code. In this paper we will use a clustering algorithm to create the error blocks. The algorithm is tuned to the variation in code and data used by the students. From these metrics conclusions will be made about the perseverance of students, and recommendations will be made to aid in the building of future LAUSD ECS syllabi.

CHAPTER 2

Data Management

Analysis and data processing in this paper were completed in R, the statistical programming language[5]. Locally weighted regression smoothing using the loess function and the R package ggplot2 were used to create the figures in the analysis [6]. The data used in the analysis was gathered from the LAUSD ECS 2013-2014 classes [7]. Use of this data was approved by the UCLA IRB.

2.1 Raw Data

The raw form of the data used in the analysis was a recording of 140 student logs from the online RStudio application used in the ECS course. Nine students had logs that contained zero lines of code, so they were not included in the data. The logs are a recording of the console output from the students programming session. Console output includes code entered, descriptive statistics, and the code generated by click through interfaces. The data consisted of two variables: code entered, and time of entry. In aggregate, 8178 lines of code were recorded from student logs. A row in this data is a single line of code entered by a student in the R programming language. Table 2.1 displays a snapshot of what a raw form of the data contained

	Code	Time
1	load(~/ecs_unit5/cdc.rda)	2014-01-10 10:42:45
2	View(cdc)	2014-01-10 10:43:01
3	table(cdc$gender)	2014-01-11 09:32:51
4	table(cdc$gender)	2014-01-11 10:44:34
5	mosaicplot(table(cdceat_salad, cdcdrink_soda))	2014-01-19 11:15:20
6	girls = subset(cdc, gender==Female)	2014-01-19 11:17:35

Table 2.1: Raw form of the Student Programming Log

2.2 Identifying Errors

Each log was cleaned of descriptive outputs and output artifacts produced by R in the compiling process. The log system used for gathering the data created timestamps for each line of code. These timestamps needed to be separated from code, so that the code could be inputted into an R console to reproduce the error a student would have received. This was done manually to ensure that the remaining information was exactly what the students had typed into their computers. This code was executed in R to test if each line of code resulted in an error. The appropriate datasets were uploaded in R to ensure that the errors would match those that the students received in the classroom. Errors from each student log were inspected manually to ensure that we were viewing the correct count of errors. A section of the ECS course relied on student gathered data, and students worked with unique datasets. Sections of the code referring to these datasets were removed because they could not be tested for errors. 1596 (19.5%) lines of code were removed from the data.

2.3 Code Characteristics

To aid in analysis, descriptive variables were formed from the information of the raw data. Four types of characteristics were identified in the characters of the code submitted by the students.

The first characteristic, **function** , identifies how many functions were specified in the single line of code. A function signifies a named section of programming that performs a specific task. In R, functions can be written by the user, however in this introductory curriculum the writing of functions is not covered. Instead, pre-written functions are taught to the students. Lines of code might contain no functions, a single function, or multiple nested functions. Examples of lines of code and the specific functions used in the line are shown in Table 2.2.

	Code	Function
1	labike$latitude	Without Function
2	girls = subset(cdc, gender==”Female”)	subset()
3	mean(table(cdc$gender))	table() & mean()
4	labike[10, 4]==”none”	Without Function

Table 2.2: Examples of Code containing a Function

The second characteristic, **access** , identifies if the code employs an operator to access specific information in a named list, vector, or data frame in the current R environment. By using an access, the student can call for subsets of an object stored in the current R workspace. Examples of lines of code with and without access operators are shown in Table 2.3.

	Code	Access Operator
1	labike$latitude	$
2	sqrt(25)	Without Access
3	table(cdc$gender)	$
4	labike[10, 4]==”none”	[,]

Table 2.3: Examples of Code containing an Access Operator

The third characteristic, **listing** , identifies if a list is within the line of code. A list specifies a grouping of multiple elements of information. Lists can be placed within functions to increase the number of parameter specifications, or used to assign multiple

values to an object. Examples of lines of code with and without listings are shown in Table 2.4.

	Code	List
1	labike$latitude	Without Listing
2	girls = subset(cdc, gender==”Female”)	Listing within subset()
3	table(cdc$gender)	Without Listing
4	labike[10, 4]==”none”	Listing within Access []

Table 2.4: Examples of Code containing a Listing

The fourth characteristic, **assignment** , identifies if the line of code stores values after the execution of the code. An assignment operator declares that a named object will store a specific set of information that is declared in the remainder of the line. Assignment in R can be completed using two operators ”<-” & ”=”. The recommended operator is ”<-”, because ”=” can also be used to declare parameter values within a function or as part of a test of inequalities. Table 2.5 shows lines of code and identifies lines containing assignment.

	Code	Assignment
1	labike$latitude <- labike$latitude - 1	<-
2	girls = subset(cdc, gender==”Female”)	=
3	table(cdc$gender)	No assignment
4	labike[10, 4]==”none”	No assignment

Table 2.5: Examples of Code containing an Assignment

Additionally, the line number for each line of code(with respect to individual student), teacher id, student id, and character length of code was identified for each observation of the data. To align our analysis with the syllabus we will remove the 8.8% of the lines that fall on non-programming days. These days occur when less than 33% of the students in a class are programming. The average number of lines produced by students on these days is five, and the lines will not be included in the analysis.

2.4 Final Dataset

The final dataset contains 5944 observations of 11 variables. Table 2.6 lists the 11 variables in the final dataset.

	Variables	Description
1	code	String of student entered code
2	errors	Dummy variable if code resulted in an error
3	time	Time that student entered line of code
4	line	Line number of code within a student's log
5	char	Character length of a line of code
6	studentid	Student I.D. number
7	teacherid	Teacher I.D. number for student's class
8	fun	Variable of the number of functions in line of code
9	access	Dummy variable if code contains an access
10	listing	Dummy variable if code contains a listing
11	assign	Dummy variable if code is an assignment line

Table 2.6: Variables of Final Dataset

CHAPTER 3

Descriptive Overview of ECS Introductory Statistical Programming Logs

The following descriptive analysis will be motivated by a search for patterns that relate to the syllabus of the course. Additionally, the data will be inspected to justify the building of content clusters of related errors, and provide insight for understanding the behavior we will see in the content clusters.

3.1 Line Count

To begin our evaluation, we will consider the code in the context of the classroom environment, so that we can understand the different factors influencing our results. A hierarchical structure is a latent context for each line of code in our aggregate dataset of lines of code. Each line of code is contained within a day, student, class, and teacher grouping. Looking at Figure 3.1, we can see that the percentage of lines that students of each teacher produced in the aggregate varies greatly. Students taught by teacher B produced 75% of the lines of code in the sample. Students of teacher C produced very few lines of code at only 7%. This allows for possible bias in aggregate statistics, because teacher B influenced a large proportion of the lines of code. If there are differences between the students of each teacher we would find skewed numbers for aggregate statistics. We will examine information from the logs to explore the null hypothesis that the students taught by the three teachers behaved in the same way.

Teacher A taught classes one and two, teacher B taught classes three and four, and teacher C taught classes five and six. Looking at Figure 3.2, we can see that there is within teacher variation in lines produced by students in each class. Students that were enrolled in class 3 produced more than 33% of the total lines of the aggregate dataset.

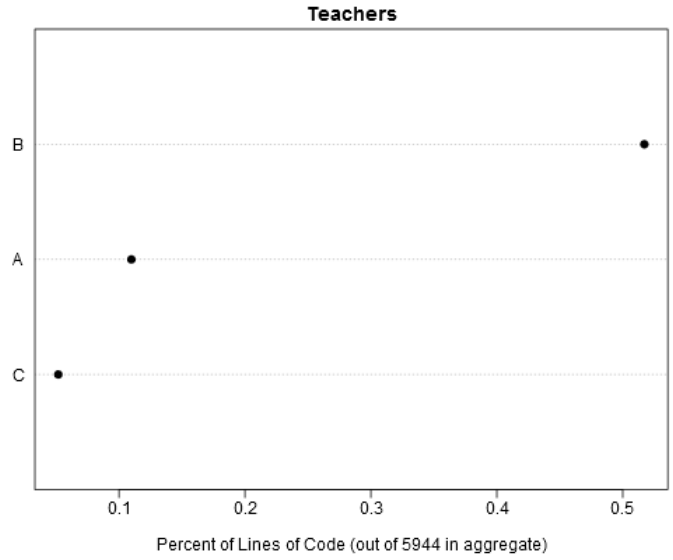

Figure 3.1: Percent of Aggregate lines of code from each Teacher

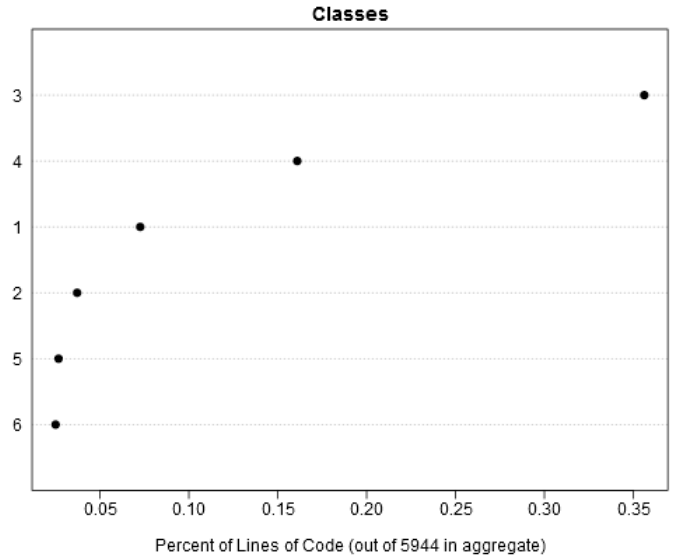

Figure 3.2: Percent of Aggregate lines of code from each Class

Looking at Figure 3.3, we can see that a large percentage of the lines in the sample were programmed on the first day of student programming. There appears to be a fairly consistent trend that later days have less representation in the sample than earlier days. Day 4 is unusual in that it is not in its expected place in the trend of lines produced. We will keep note of this difference when inspecting future plots. Looking at Figure 3.4 we can see that the number of lines produced by each student varies greatly. A small subset of students produced a large number of lines. These students who produced a

large number of lines of code will have a larger influence on aggregate statistics that we view.

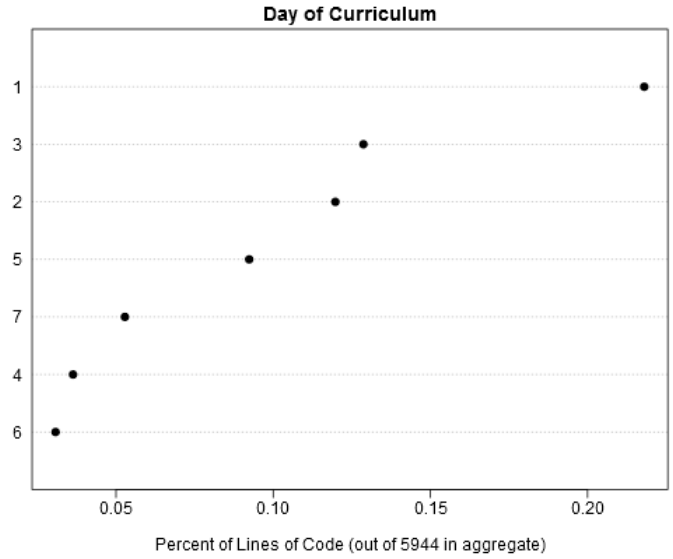

Figure 3.3: Percent of Aggregate lines of code from each day of curriculum

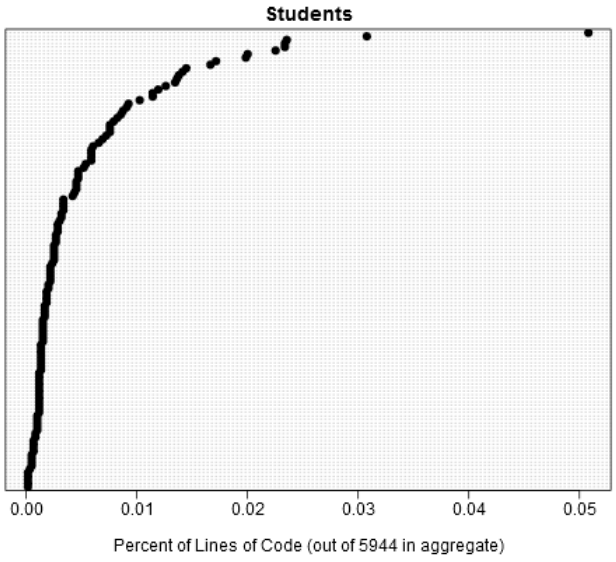

Figure 3.4: Percent of Aggregate lines of code from each Student

It would be desirable to establish a standardized method of observation before delving into further descriptive statistics. Because this ECS course has a syllabus and daily instruction plan, a standardization by day of curriculum would be preferred. Figure 3.1 and 3.2 revealed that there is imbalance in line count among the teachers and classrooms. Figure 3.5 shows that the number of students who stop programming is fairly proportional

to the remaining number of students still programming. For this reason, our finding that day one had a much larger proportion of lines is logical. This rate results in very few lines, and students (less that 5%) being observed for days eight, nine, and ten. Looking at Figure 3.6 we can see that up until day eight the average number of lines produced by individual students each day stays fairly constant. So for consistency, most of the following graphics will be displaying information against the day of curriculum it was produced, and only include the first seven days of curriculum.

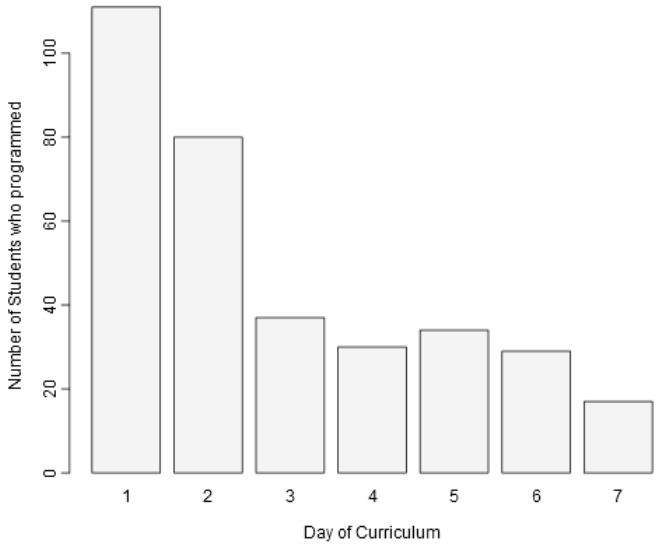

Figure 3.5: Number of students who programmed on each day of curriculum

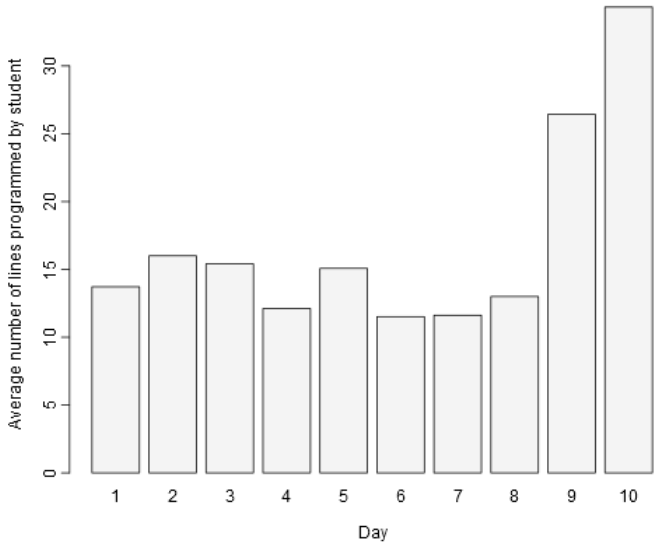

Figure 3.6: Average number of lines per student on each day of curriculum

Figure 3.7 shows the variation in the number of lines programmed by students on each

day of curriculum. Within day variation is greater than between day variation. For this reason, we should include variation in our observations to support our understanding. We can see that the median number of lines produced by students on one day is consistently lower than the mean number of lines on the same day. Some students produced a much larger amount of lines on a single day than the average/median number of the day. We will move forward with looking at aggregate statistics by the day of curriculum on which the lines were produced. We will focus on days one through seven as to avoid over fitting our findings to the few students who programmed on days eight, nine, and ten. We pay attention to day four as it has a very low number of lines of code. The current syllabus plans for five days of programming by each student before they begin their final project, so the seven days for observation should capture most of the syllabus driven programming.

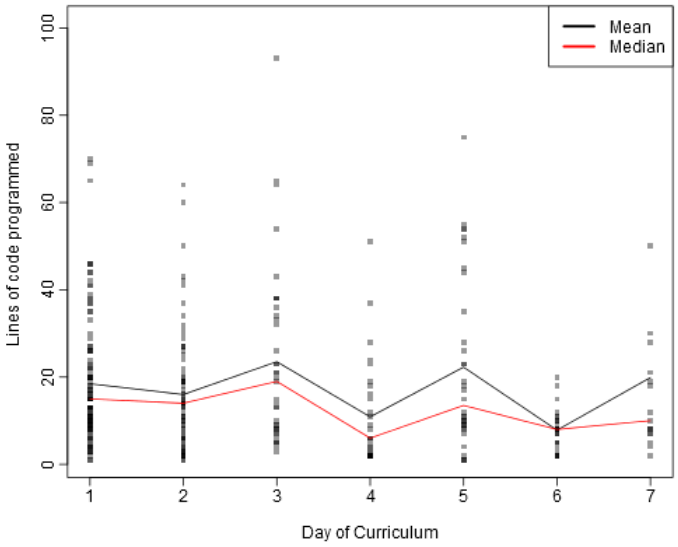

Figure 3.7: Distribution of lines of code programmed by students on each day of curriculum

3.2 Teachers and Classes

The three teachers divide the students into three groups. As previously mentioned, each teacher was responsible for two classes of students. We are interested to see if the teachers are associated with different behaviors in their students. Additionally, we are interested to see if there are differences between the two classes taught by the same teacher. Understanding how student behavior was different will provide information for

comparison of lines between students from different classes. Looking at Table 3.1 we observe the number of students programming on each day. The pairs of classes taught by each teacher show similar patterns of student programming on the same days. Of particular concern, the classes taught by teacher C participated in programming on only one day. If we wish to make conclusions about learning, it would be difficult to measure within one day. For the evaluation in this paper, the measurement of learning will be limited to classes one through four.

Day :	1	2	3	4	5	6	7	8	9	10
Class 1	30	22	6	4	0	0	0	0	0	0
Class 2	14	11	7	4	2	1	1	0	0	0
Class 3	33	33	31	28	27	17	14	7	3	2
Class 4	20	20	20	15	12	9	6	4	4	1
Class 5	21	0	0	0	0	0	0	0	0	0
Class 6	13	0	0	0	0	0	0	0	0	0

Table 3.1: Number of students programming on a day of curriculum by Class

Table 3.2 provides a breakdown of the mean value of characteristic statistics. The value shown is the mean of student values. We can see that for the most part the pattern of similar results for the pairs of classes taught by each teacher is continued here. There are also trends that appear to be similar for all classes. For example, it appears that the mean number of functions in a line of code is close to one, so function use is a very common tool used by students across all classes. There are some early indicators of differences between classes in this table such as the fact that teacher C did not have a single student program a line with an assignment operator. We will next view these same characteristics by day of curriculum to see if there are additional patterns within the contents of the code.

	Line	Day	Function	Access	Listing	Assign	Characters
Aggregate	49.77	3.40	0.99	0.41	0.31	0.05	32.97
Teacher A	26.52	2.50	1.02	0.35	0.38	0.09	30.23
Teacher B	92.98	5.70	1.05	0.42	0.40	0.06	44.22
Teacher C	12.50	1.00	0.83	0.48	0.06	0.00	18.99
Class 1	25.17	2.37	1.03	0.35	0.40	0.10	30.29
Class 2	29.43	2.79	1.02	0.36	0.35	0.08	30.11
Class 3	98.30	5.82	1.01	0.51	0.39	0.05	41.06
Class 4	84.20	5.50	1.12	0.26	0.42	0.07	49.43
Class 5	11.52	1.00	1.01	0.50	0.10	0.00	19.40
Class 6	14.08	1.00	0.55	0.46	0.00	0.00	18.32

Table 3.2: Mean of student characteristics - Aggregate/Teacher/Class

We will start our evaluation by viewing the number of characters that were input into lines of code. Looking at figure 3.8 it appears that an increase in the number of days of programming is associated with an increase in the number of characters in a line of code. If we look at figure 3.9 we see that this pattern is mostly driven by the influence of the students taught by teacher B shown in yellow. There is not an average line for the students taught by teacher C because there was only entries on one day for all students of teacher C. It appears that the students taught by teacher A show a fairly constant average length of code, while those of teacher B show an average that increases by day. Looking at Figure 3.10 we can see that the average length of code of the two classes each teacher taught are very similar on each day.

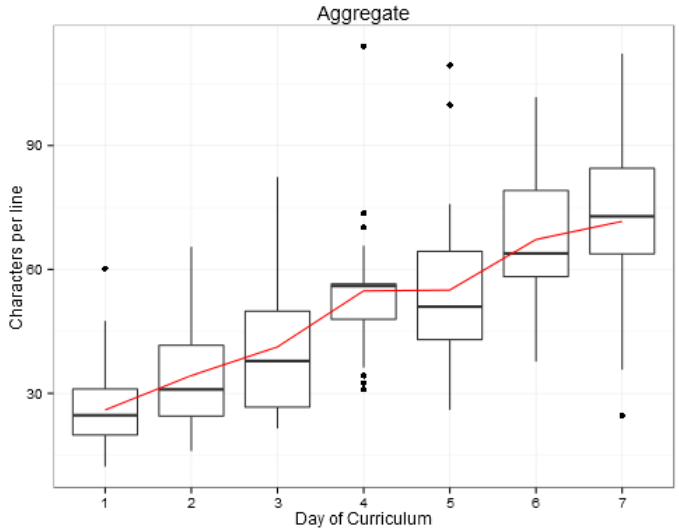

Figure 3.8: Code character length by day of curriculum - Aggregate

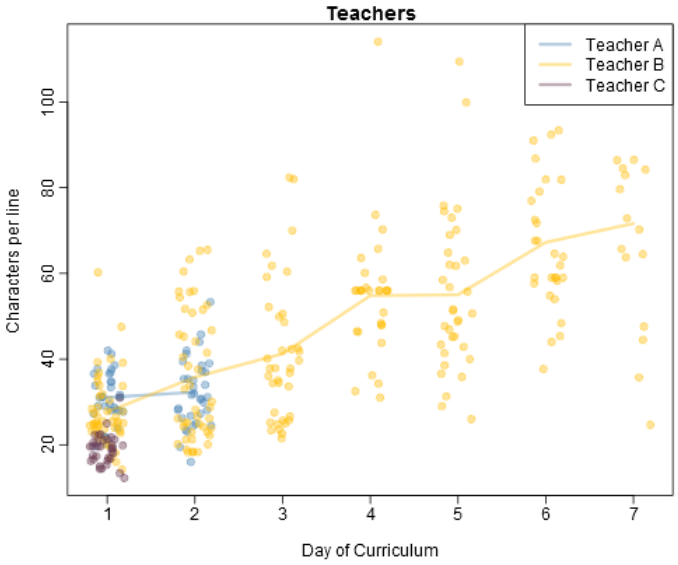

Figure 3.9: Code character length by day of curriculum - Teachers

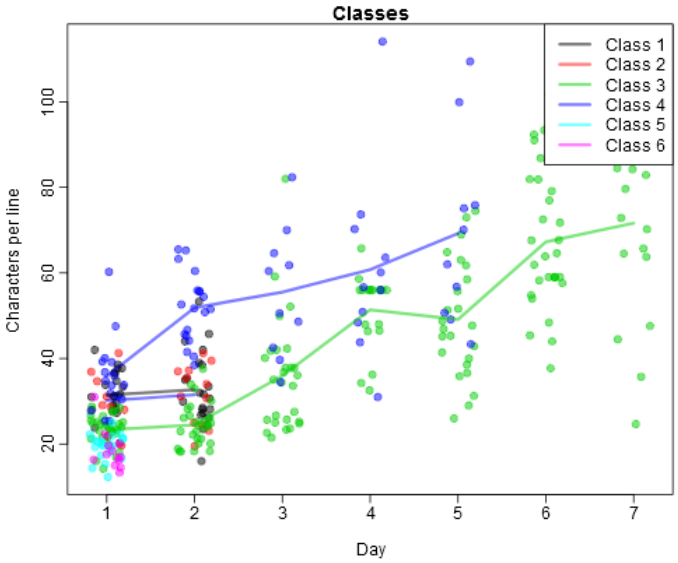

Figure 3.10: Code character length by day of curriculum - Classes

With the knowledge that average characters differ between classes we can expect a difference in the characteristic content of the lines of code. Looking at figure 3.11 we can see that an increase in average number of functions in a line of code is associated with an increase in day. On day one, just under one function per line is used. As we approach day seven, the average is 1.3 meaning that multiple functions in a single line becomes more common. Looking at figure 3.12 we see that students taught by teacher B show an increase from just under one function per line to the before mentioned 1.3 average. Students taught by teacher A show a more constant rate of function use hovering around 1 on days one and two. Figure 3.13 shows that the classes taught by the same teacher showed similar function use patterns. It is starting to become apparent that the patterns of the two classes taught by the same teacher are similar.

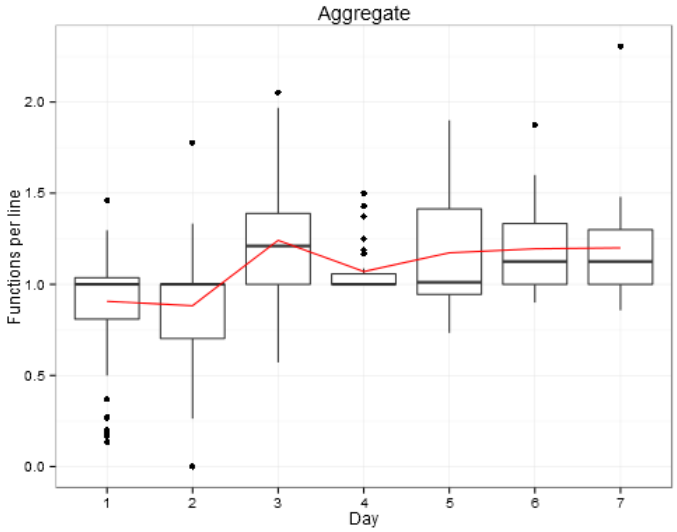

Figure 3.11: Function count per line by day of curriculum - Aggregate

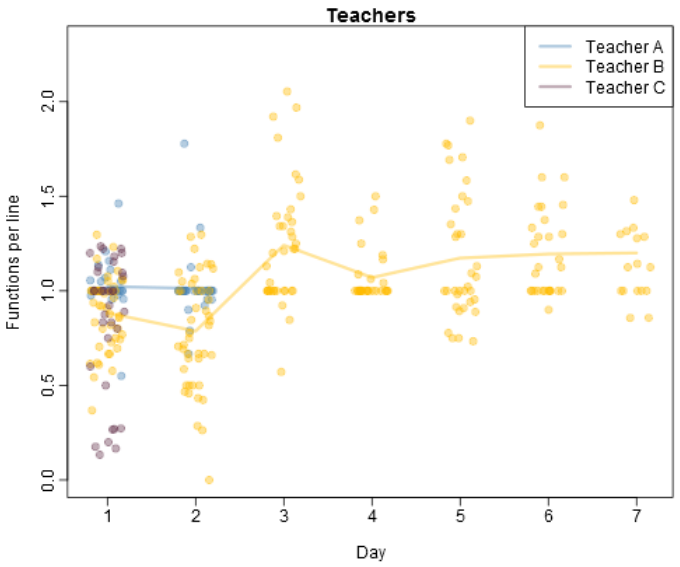

Figure 3.12: Function count per line by day of curriculum - Teachers

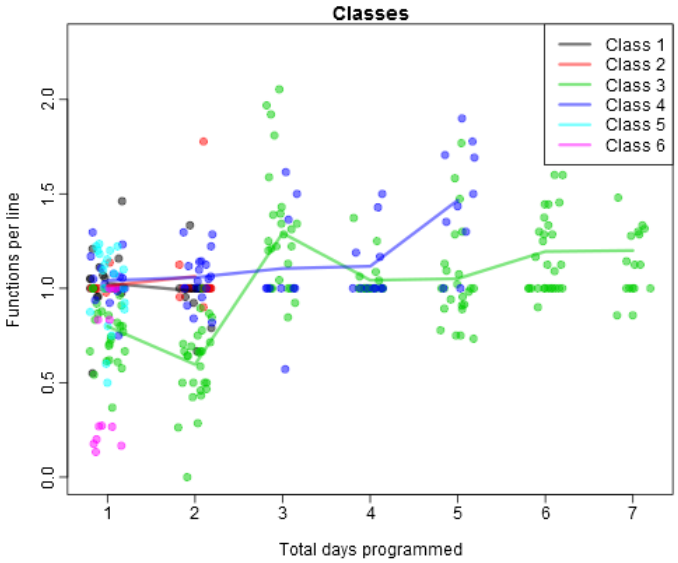

Figure 3.13: Function count per line by day of curriculum - Classes

Looking at Figure 3.14 it appears that the percentile range of access operator usage is very large relative to the bounded values of 0 to 1 that the variable can take. Looking at Figure 3.15 we can see that there is a large variation in the number of access operators used by students in the same class and in different classes. Figure 3.16 reveals the same variability, and that the classes taught by the same teacher do not appear to have similar values when using this visualization. If there is a pattern in access we will need to employ further information to make conclusions.

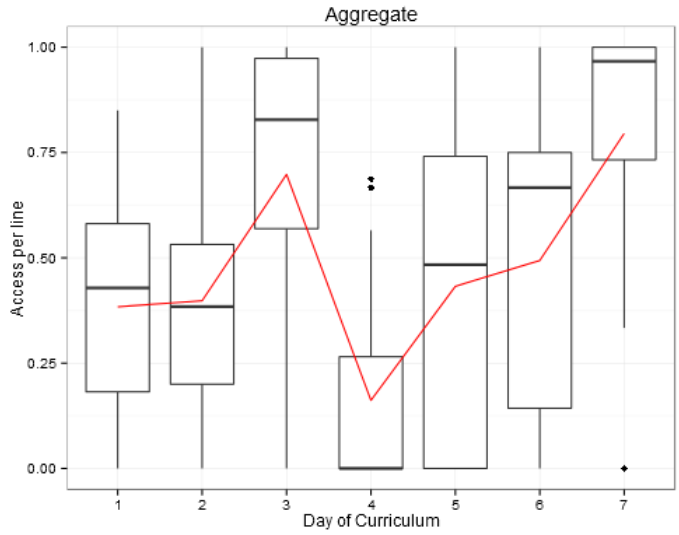

Figure 3.14: Access count per line by day of curriculum - Aggregate

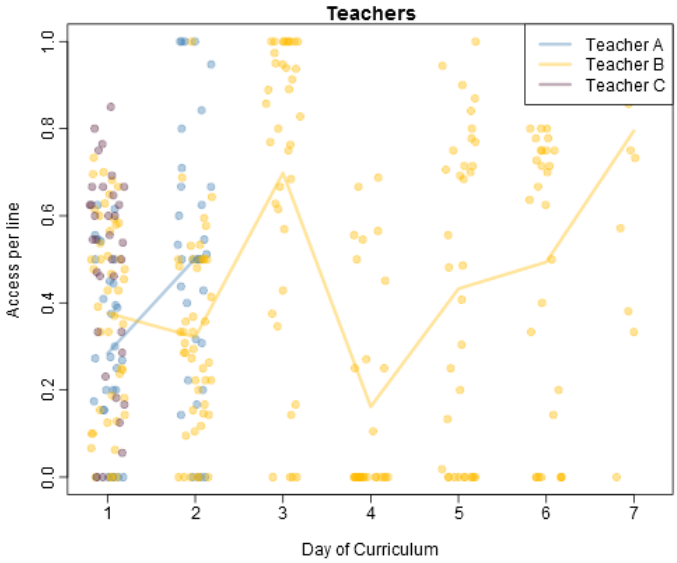

Figure 3.15: Access count per line by day of curriculum - Teachers

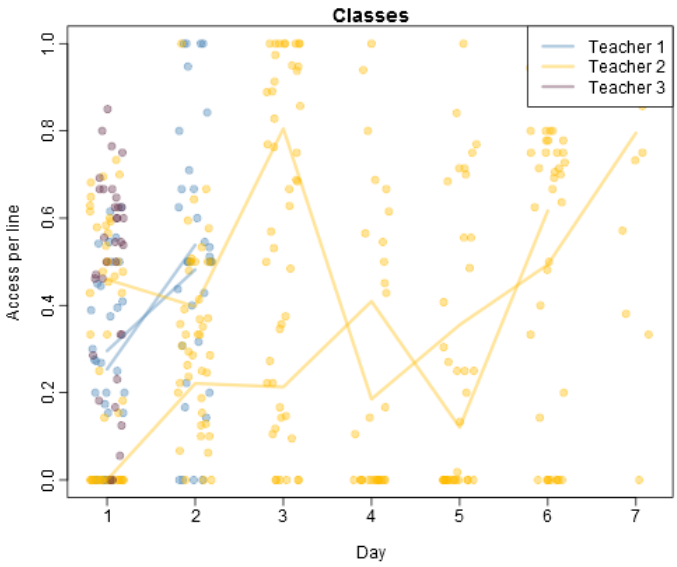

Figure 3.16: Access count per line by day of curriculum - Classes

Looking at the average line in Figure 3.17, we see that the average number of listings per line increases as day of curriculum increases. Figure 3.18 highlights that there is not a clear difference between classes one through four in listing usage. Figure 3.19 also shows that the four classes have an overlapping pattern of listing usage. It appears that in the later days, students who used listings began to use listings frequently. So we see very few students with a low usage rate of listings.

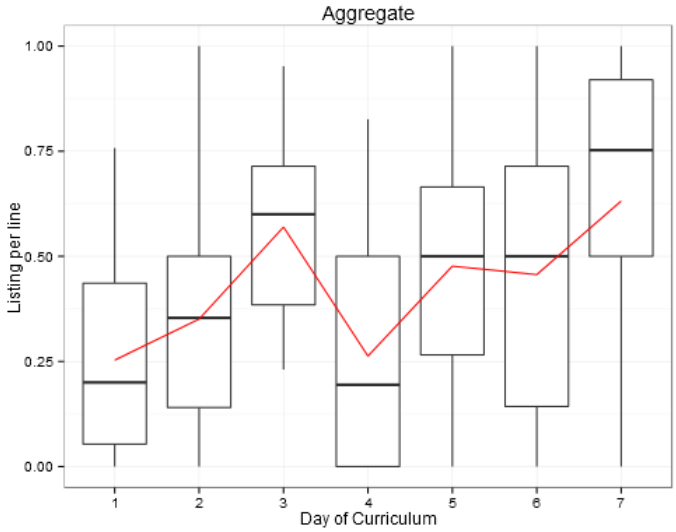

Figure 3.17: Listing count per line by day of curriculum - Aggregate

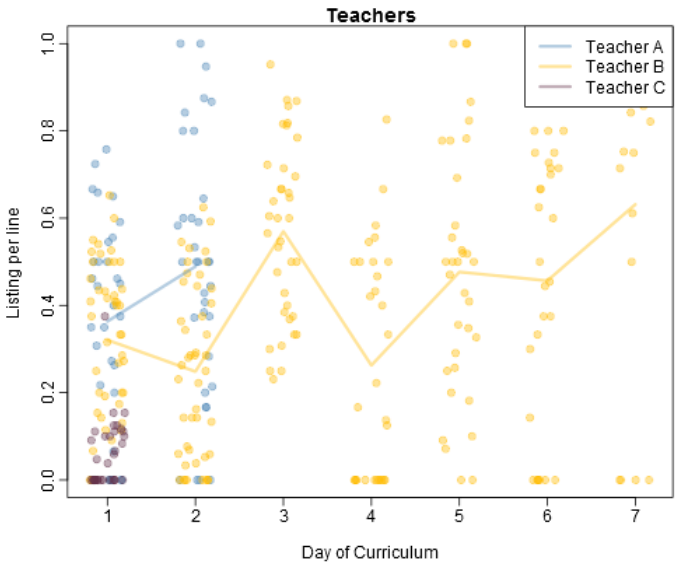

Figure 3.18: Listing count per line by day of curriculum - Teachers

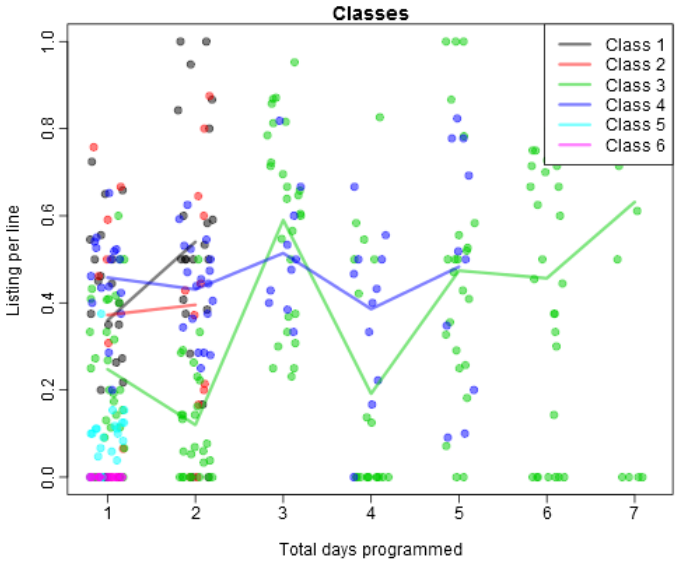

Figure 3.19: Listing count per line by day of curriculum - Classes

The last characteristic to view is assignment operator usage per line. Looking at Figure 3.20 we can see that the rate of assignment operators is much lower than the other characteristics. The highest rate of assignment is .5, or an assignment in half of a student's daily lines. Figure 3.21 reveals that usage of assignment operators decreases with number of days for students in classes one and two. Further inspection of the code is needed to understand why this might be happening. The large number of zeros makes it challenging to view trends in this distribution of assignment. Figure 3.22 shows that the continued patterned of similarities of characteristic usage in paired classes is also shown with assignment operators.

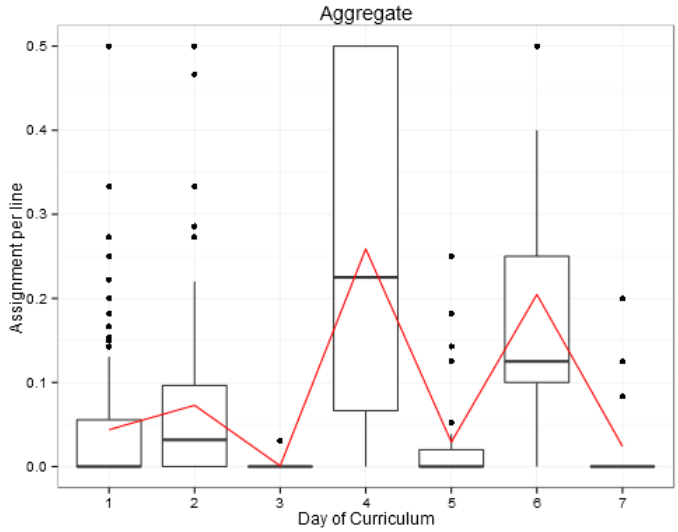

Figure 3.20: Assignment operator count per line by day of curriculum - Aggregate

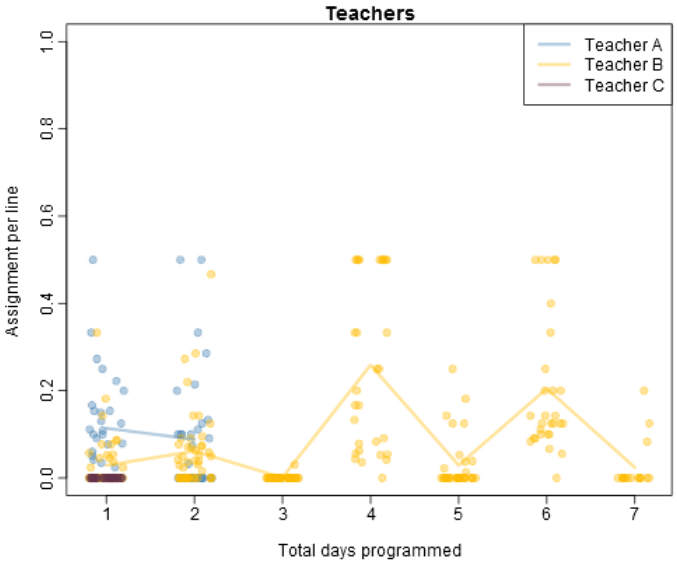

Figure 3.21: Assignment operator count per line by day of curriculum - Teachers

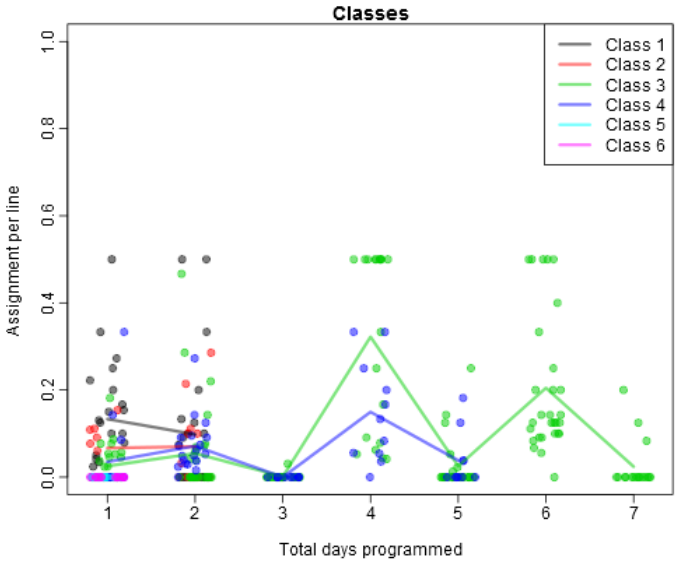

Figure 3.22: Assignment operator count per line by day of curriculum - Classes

We have seen that the averages of classes taught by the same teacher appear to be similar. Overall function usage and listing usage appear to increase across all classes. The patterns of assignment and access are not as clear, so we will view the content of the lines of code to look for further understanding in these patterns.

3.3 Code Contents

The content of the individual lines of code allows for observation of what the students entered into their Rstudio console on days of programming. The students were given guidance into what their goals for a specific task were, and the recommended functions to use for this goal. There is opportunity for students to enter only the required lines, explore code, make spelling mistakes, and even enter non-sensical writing in the console of Rstudio. Examining these lines of code can give us insight into student behavior while programming.

The scripting environment that the curriculum of the ECS class is based on relies on the usage of pre-written functions. Table 3.3 highlights the most used functions in the sample.

At the top of the table, the most used function is View(). View() is an interesting function to observe when using Rstudio because it does not have to be typed into the

Function	Count	%
View	1568	0.24
table	1214	0.19
plot	951	0.15
subset	908	0.14
barplot	666	0.10
MakeMap	240	0.04
load	231	0.04
read.csv	211	0.03
mosaicplot	190	0.03
hist	150	0.02
c	148	0.02
summary	128	0.02
boxplot	84	0.01
head	67	0.01

Table 3.3: Most Used Functions

console. When using the Rstudio interface, if the user clicks on the name of a data set in the Rstudio workspace then the function View() is executed in the console. With 24% of the lines containing the View() function, we can know that the students are looking at the data, but it is difficult to say how often students are clicking on the interfaces versus typing in the function.

Another interesting function in this list is load(). This function was used in 4% of the lines and just as frequently as the read.csv() function. These two functions are the main functions used by the students to upload data into the Rstudio software. When looking at load(), we should be aware that if a student used load to bring in their data, they would not have used an assignment operator. A student who uses read.csv() instead, would have used an assignment operator. This difference may explain the unusual patterns that we saw in the assignment operator plots of the previous section. Another explanation for the decrease in assignment operator is the online Rstudio application. This version of Rstudio maintains the workspace between days, so the students do not have to reload

the data each day and use assignment operators in the process.

Table 3.4, 3.5, and 3.6 show the five most used functions by teacher. We see that the functions are similar for all three teachers. They include View(), some form of data manipulation, and plotting functions. It appears that teacher B did not have their students use the load() command. This might explain the difference between teachers in the assignment operator usage graph.

Function	Count	%
View	303	0.26
MakeMap	237	0.20
subset	202	0.17
plot	104	0.09
load	72	0.06

Table 3.4: Most used functions - Teacher A

Function	Count	%
View	1158	0.23
table	1078	0.22
plot	766	0.16
subset	706	0.14
barplot	614	0.12

Table 3.5: Most used functions - Teacher B

Function	Count	%
View	107	0.25
table	85	0.20
plot	81	0.19
load	71	0.17
barplot	40	0.09

Table 3.6: Most used functions - Teacher C

Table 3.7 shows us the pattern of data set usage on the main days of programming (Days where at least 33% of the class programmed). It appears that the most used data set was cdc. The cdc dataset and labike dataset were used primarily on the first two days of programming. Day four and beyond show primarily usage of the two Mobilize data sets. It is interesting to see that there were 394 lines of code that used the same data set called "Bullied". Bullied was a subset of the Mobilize_SchoolSafety_2011 data set. This name was the name that teacher B used when instructing their two classes. It is interesting to see that the students utilized the same name as their teacher when they had the option to choose their own name for the new data frame they created from Mobilize_SchoolSafety_2011.

Data Set	Lines
cdc	2187
Mobilize_ExerciseHealth_2011	904
Mobilize_SchoolSafety_2011	764
labike	568
Bullied	394

Table 3.7: Most used Data sets - Aggregate

We have found that the function usage and data set usage differed with day and class. This begins to give us a more granular view of what is happening when the students program in the logs. We can see that the characteristic variability may be related to content variability and teacher choice of function and data set. Next we will look at how the time of code entry varies in the logs.

3.4 Line of Code Entry Time

Along with the code that was entered, the logs recorded what time the students entered the code into the R console. This time allows us to build a timeline of when the code was entered. We will make observations of student behavior based on these times. To begin our evaluation of time we will look at what time during the class period the students entered the code. For clarification we will define two terms:

Time of line of code entry The number of minutes that have passed since the beginning of the current day of curriculum when a line of code is input in the R console.

Time between entry The number of seconds between a line of code and the next line of code input by the same student in the R console. This time difference is not recorded for lines of code on different days of curriculum.

Figure 3.23 is the distribution Time of line of code entry for the two classes of teacher A. Looking at Figure 3.23, it appears that the middle of the class is the time when students entered the most lines of code into their logs. The peak time of code entry is around 40 minutes into the class period. It appears that students were given to opportunity to enter code after the one hour mark in this classroom. With some of the entries occuring after 84 minutes.

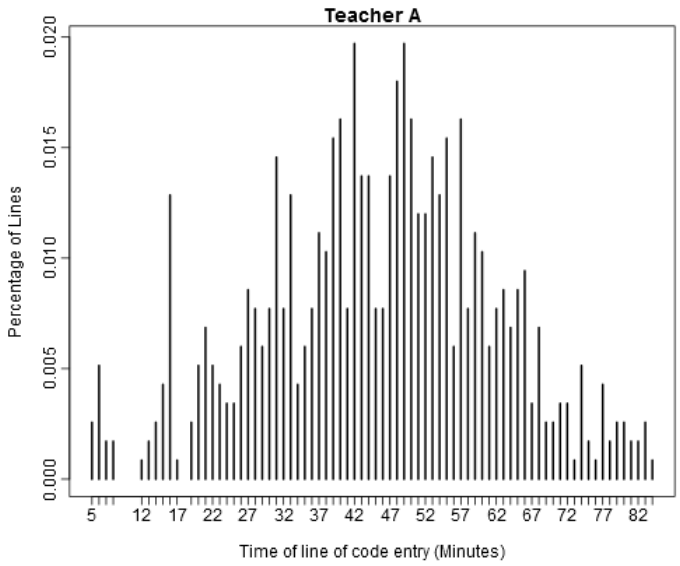

Figure 3.23: Time of line of code entry - Teacher A

Looking at Figure 3.24, we can see a very similar distribution of Time of line of code entry. The students in the classes of teacher B entered the majority of their lines during the first hour and twenty minutes of the class. Figure 3.24 also shows a pattern of a few lines of code being programmed after this time. This would suggest that the times presented for the students to program either differed from that of teacher A, or students stayed after class to finish programming. Looking at Figure 3.25, we see that the low number of lines from teacher C's students have not converged to an apparent trend in Time of line of code entry.

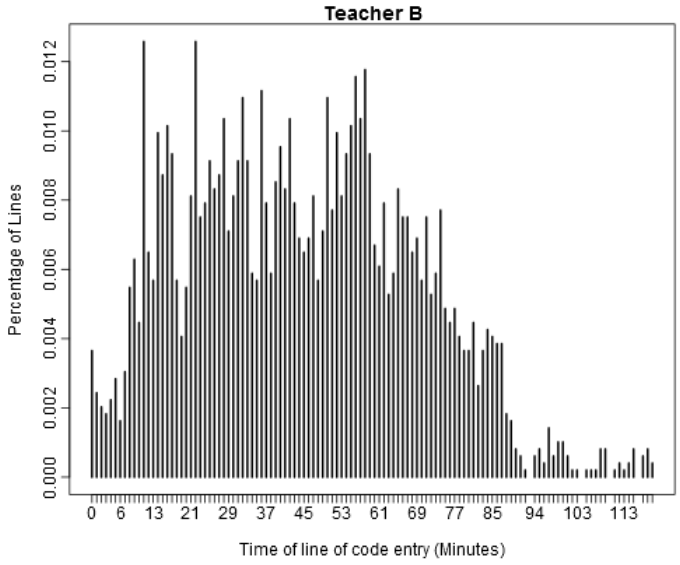

Figure 3.24: Time of line of code entry - Teacher B

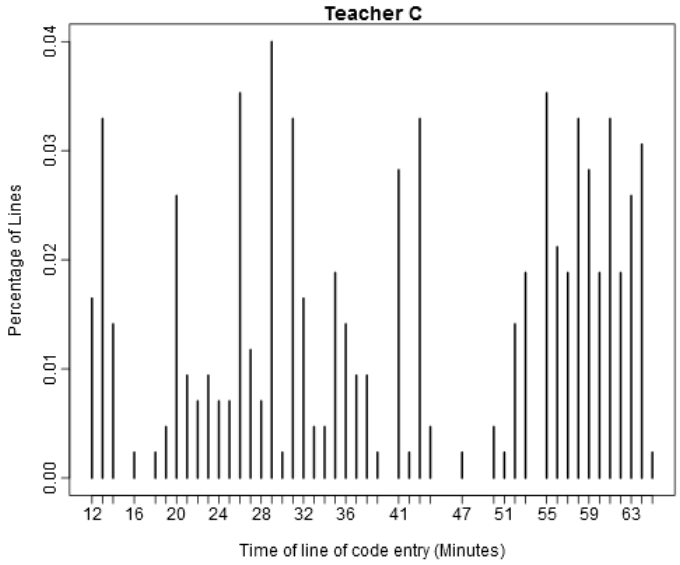

Figure 3.25: Time of line of code entry - Teacher C

Looking at Table 3.8, we can see that the majority of lines are entered within one minute of the previous line.

Percentile:	25	50	75	90	95	98	99
Seconds:	10	35	119	358	650	1128	1583

Table 3.8: Percentiles of time between entry - Without View()

Figure 3.26 reveals that the distribution of time between entry is very much right skewed. This means that it is possible that there is clustering happening for the times of entry. Figure 3.27 shows that the distribution of time between entry is approximately log-normal. There is an unusual amount of lines with a difference very close to log(0). With further inspection of these lines it appears that the function View() is responsible for this pattern. The View() function is called immediately after some instances of uploading data.

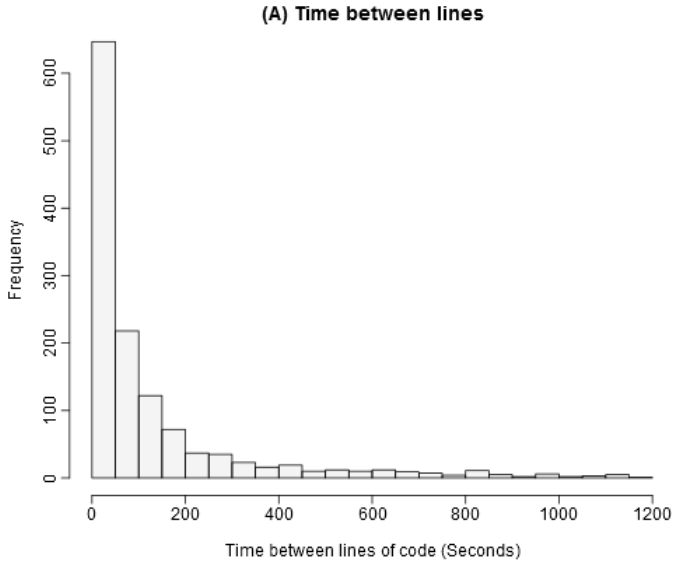

Figure 3.26: Time between entry - Aggregate

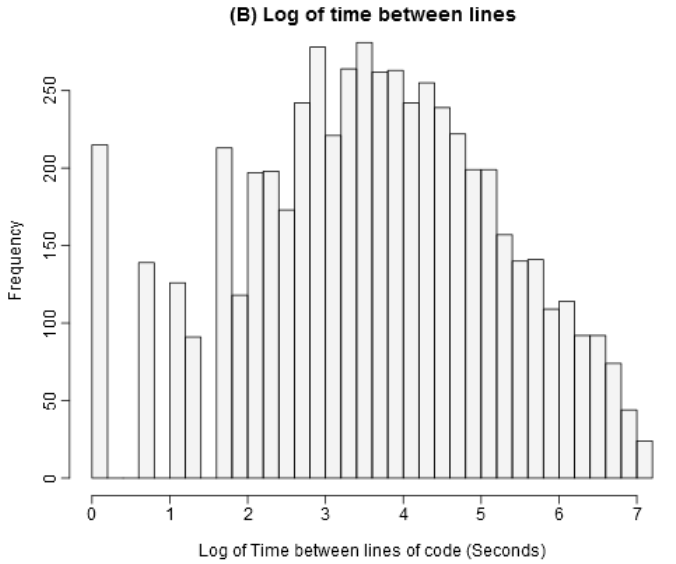

Figure 3.27: Log time between entry - Aggregate

We can remove the lines of code that had an automatic and immediate entry into the console due to the View() function behavior from our calculations. Table 3.9 shows that more than half of the lines of code are still entered within one minute of the previous line.

Percentile:	25	50	75	90	95	98	99
Seconds:	20	54	160	440	751	1228	1751

Table 3.9: Percentiles of time between entry - Without View()

One possible cause of the time between entry is the length of code. Lines of code with a larger character length would take longer to type in and would change entry times. Looking at Figure 3.28, it appears that character length of code is not strongly associated with the time between entry. The correlation for the association is .13. This suggests that something outside of typing is influencing the time between entry of lines of code.

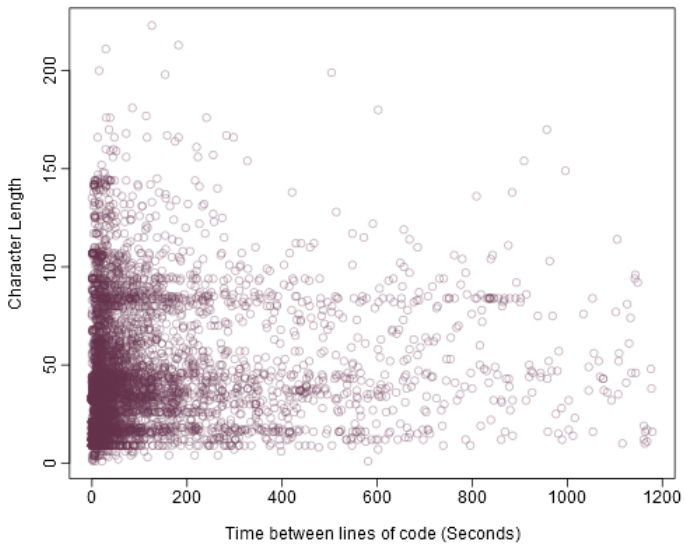

Figure 3.28: Code character length by time between line of code entry

From looking at the time of entry for lines of code, we have seen that there are patterns in the entry time. It appears that the students are entering the most lines of code during the middle of the class period. We have also shown that more than half of lines are entered within one minute of the previous line. The View() function accounts for automatic entries into our logs, and with the View() function removed the distribution of

time between lines is still heavily right skewed. When a log transformation is applied to the distribution of time between entry of code we see that the times are approximately log-normal. This suggests that there may be multiple independent variables driving the time between entry. We tested the length of characters in the lines of code to see if it is the main driver for the variation, but it does not appear to be. This leaves open other factors to explain the time between lines of code such as time within the class period, lesson structure, confusion with the assignment, and number of lines needed to complete a cluster of data analysis.

3.5 Errors

One measure that is used in programming assessment is program errors. Because program error would normally describe the failure of a chunk of multiple lines of code to perform a specific task, our definition for single line error will need to have a different interpretation than that which may be used for other courses. Our definitions will be as follows:

Error When a single line of code is input into the R console and returns an error message after compilation.

Error Rate The proportion of lines of code that return an error. For the aggregate of all lines of code the error rate was 27.82%.

Statistical programming in the ECS course is done with the purpose of understanding a data set. Errors in code would hinder speed of data analysis. Consecutive errors may result in failure of a specific analysis task due to lack of programming ability. This failure of analysis would more closely mirror a program error from general programming courses. To begin the evaluation of error rate we will first look at class-wide error rates. A reduction in error rate as the class progresses would suggest that students have reduced their number of mistakes in individual lines of programming during their data analysis. Looking at Figure 3.29 we see that further into students' programming logs the error rate slightly increases when looking at the error rate for the aggregate of all students. One possible explanation for this increase in error rate is complexity of code.

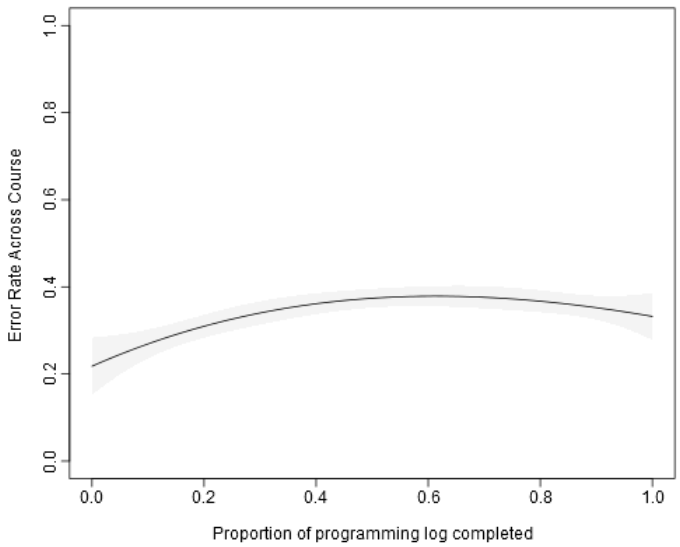

Figure 3.29: Aggregate error rate by proportion of programming log completed

Instead of viewing error rate over percentage of programming log completion, we can observe how the error rate changes as the students complete lines of code. Looking at Figure 3.30, Error rates across number of lines completed shows variability as the number of lines completed increases.

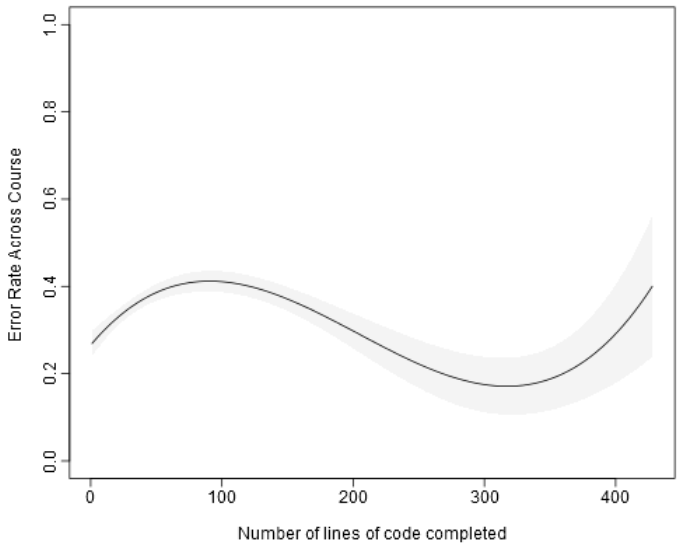

Figure 3.30: Aggregate error rate by number of lines completed

Additionally, Figure 3.31 shows that error rate shows a slight increase as the days of curriculum increase.

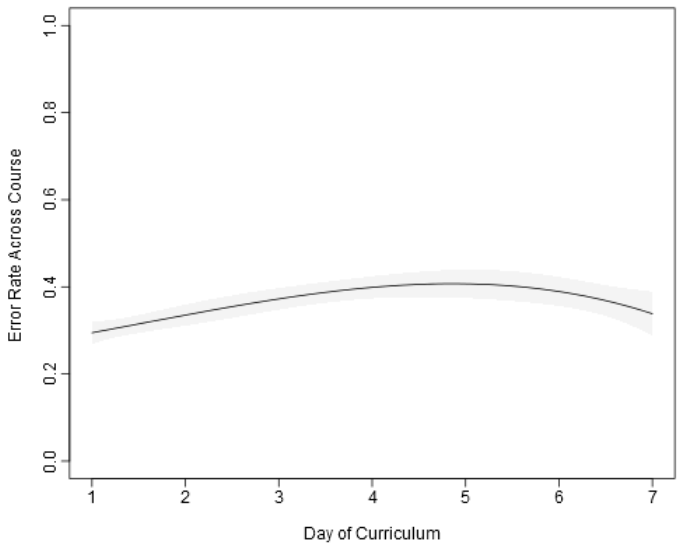

Figure 3.31: Aggregate error rate by day of curriculum

Each of these plots focuses on the frequency of errors among entered lines of code across the aggregate of lines. With the patterns we have found in our observation of the descriptive statistics, it would be difficult to assume that the lines of code are perfectly independent. To fully understand and interpret the errors there would need to be more granularity than a aggregate view. We will now explore the distribution of errors to see if we can find motivation for a clustering of errors to provide granularity while also maintain context for the lines.

Looking at Table 3.10, we can see that the time between errors is quite varied. Twenty-five percent of errors occur within 22 seconds of the previous error. However, more than 5% of the errors are thirty minutes after the previous error. We should hesitate to assume that the relationship between errors 22 seconds apart and 30 minutes apart are the same.

Percentile:	25	50	75	90	95	98	99
Seconds:	22	74	272	828	1338	2189	2785

Table 3.10: Percentiles of time between errors - Without View()

We have plotted the distribution of time between entry of errors in Figure 3.32. The distribution is right skewed and shows that most errors occur in close time intervals of another error. There may be some dependence between the errors that is causing this

skew in the distribution.

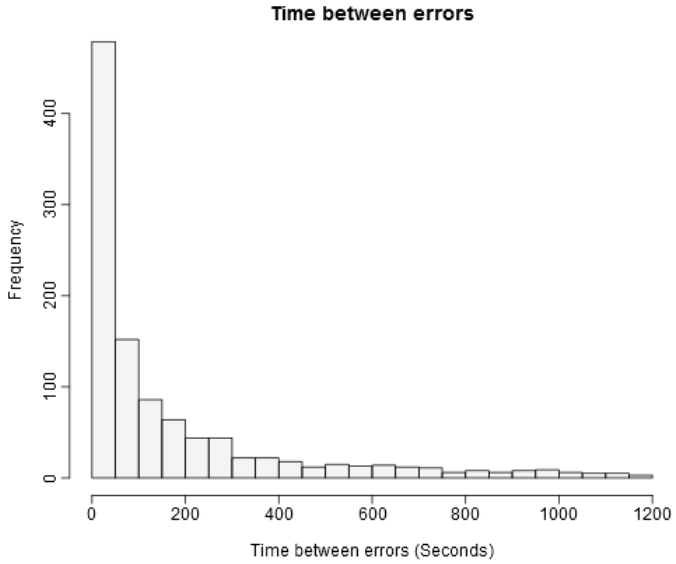

Figure 3.32: Time Between entry of lines of code with Errors

As additional information for understanding the dependence between errors, we see that there is a 26% error rate when the previous line of code was not an error, but there is a 52% error rate when the previous line of code was an error. This doubling of the error rate suggests that there is a relationship between errors.

3.6 Descriptive Overview Conclusions

Our examination of the descriptive variables has led us to an interesting idea of clustering errors. We have found that the lines of code are unequally representing the different levels of the hierarchical context of the lines of code. This makes it difficult to explore the individual lines of code without providing context. We saw that the characteristics of the lines of code entered vary by teacher, classroom, and day. This makes it difficult to compare individual lines of code without paying attention to their contents. It was shown that the distribution of time of line of code entry varies by time during the class period. Additionally, the time between entry is not the direct result of the length of the code that needs to be typed into the console. This suggests that we have not recorded all of the drivers of time between lines. Factors such as thought, and reading through notes may be influencing the time between entry. We did not uncover information to

discourage us from thinking that the errors might be clustered. For these reasons, we will construct clusters that provide context for lines of code to offer another means of evaluation.

CHAPTER 4

Formative Assessment of Errors via Content Clustering of Lines of Code

To begin the section we will define a set of terms relevant to our analysis.

Content Clustering A grouping of information into clusters based on the content of the information.

Correction Rate Proportion of error blocks in a subset that end when a student successfully corrects the initial error.

End of Block The error block ends when a student either successfully corrects the initial error, or stops entering lines of code that attempt to correct the error.

Error Block A content cluster that contains a line of code with an initial error, and any additional lines of code with attempts made at correcting the initial error.

Formative assessment The evaluation of the relationship between the student, the teacher, and the learning environment during the learning process. In this paper, this will be the interaction between the student and the error responses they received while programming in R. This will be measured by perseverance.

No Correction Failing to correct an error in the current error block due to time, change in code to a different task, or change in day of curriculum.

Perseverance The number of attempts made to correct an error, or the amount of time needed to correct an error.

Persevere Attempting to produce the correct response after receiving an error message.

Successful Correction Entry of an error free line of code that corrects the initial error of the current error block.

Summative Assessment Assessment which occurs at the end of a learning timeline. A common example of summative assessment is final exams.

Time between attempts This is the same as time between entry. The time(in seconds) between the entry of one line of code, and the entry of a proceeding line of code that is in the same error block.

Time between errors The time(in seconds) between the entry of a line of code that returns an error, and the entry of the next line of code that returns an error during the same day of curriculum.

Table 4.1 shows an example of an actual error block from a students log that contains a successful correction. The time between entry is shown in the table and highlights the non-uniformity of the time between entry that is found in error blocks. Viewing the code contents shown in Table 4.1 one can see how the student made small alterations to the previous line of code in an attempt to fix an error. This particular error block is much longer than average, but shows the progression students often made while dealing with an error.

Attempt	Code	Error	Time Between
1	plot(long$lat)	Yes	0
2	plot(bus_stops$long, lat)	Yes	51.00
3	plot(bus_stops$long,lat)	Yes	53.00
4	make plot(bus_stops$long, lat)	Yes	44.00
5	MakePlot(bus_stops$long, lat)	Yes	53.00
6	Plot(bus_stops$long, lat)	Yes	8.00
7	Plot (bus_stops$long, lat)	Yes	118.00
8	plot (bus_stops$long, lat)	Yes	8.00
9	plot (bus_stops$longitude,latitude)	Yes	34.00
10	plot (bus_stops$lat, long)	Yes	19.00
11	plot(bus_stops$lat, long)	Yes	21.00
12	plot(bus_stops$lat,long)	Yes	10.00
13	plot(bus_stopslat, bus_stopslong)	No	17.00

Table 4.1: Example of a Cluster

The R environment has a built in feedback system of errors when students input their code into the R console. The error messages that students receive are a type of formative assessment. The formative assessment type we will investigate in this paper is when feedback moves learners forward during learning[9]. Students have the opportunity to correct a mistake based on the information in the feedback, and move forward in their analysis with a new corrected line of code. In order to correct the code, a student would have to be able to notice the mistake in their previous line of code, or be able to use resources to find the correction. The focus of this paper is not how to improve this feedback, but instead to understand the way students interact with the feedback they receive. We will consider a student attempting to input a correct line of code, after they received an error message, as an example of perseverance. In this way, we will be able to evaluate learning by measuring how students persevere when receiving errors messages.

Let us consider the interaction that a student in an introductory statistical programming course would have with the formative feedback system of errors. When a student enters a line of code, they would have an expectation of what the compilation of the code

would cause. But when met with an error message, the student knows that something occurred that they did not expect. Entry of a new line of code that corrects the error that was just produced would show that the student was able to find the correction in some manner. Evaluating what happens between the initial error and the final correction, or lack thereof, will allow us the understand the students interaction with the formative assessment. While correct lines of code measure success of the compilation, the ability to identify and correct errors is a valuable skill when learning how to program. Comparing how the students interact with the error messages on successive days will allows us to see if the students are learning.

To provide context, we organized the lines of code in the logs into error-based content clusters that we will call error blocks. In this structure it may be possible for multiple lines of transition errors to fall between the initial error and the end of block. In a sense, we are forming the cluster to view if a student was able to successfully correct an error or if they moved on to other code without a correction. We will assume that, on average, clusters that end in a successful correction of the error display a more desirable outcome than clusters that do not lead to correction. Also, we will view error blocks that had more time entering codes before failing to correct the error as evidence of higher Perseverance . Through the logs, we have access to when errors were given as feedback, and the time between lines of code. We will measure the frequency of successful corrections, time until correction or failure, and number of lines of code (attempts) entered in the error block. Improvement in metrics related to these blocks would be associated with learning, so we would have a way of loosely approximating learning related to the formative assessment.

4.1 Algorithm for Formation of the Clusters

To identify the Error blocks in the LAUSD ECS logs, we used an algorithm that was tuned to the specific variation in functions and data sets. For future use of a similar algorithm, the number of functions and data sets would need to be considered when deciding cut-off values for the algorithm.

There are two pieces of information that are compared to build the error block. First, the outcome of a line of code code is determined to be either an error, or non-error.

Second, the line is inspected to determine if it is connected to the current error block. Lines are inspected sequentially by time of entry in the log. The initial error for an error block is determined by order of inspection. So the first error in the log after the completion of the previous error block is the initial error of the new error block. Error blocks are limited to within a single day. The View() function is omitted from the logs because of the interaction of clickable interface in Rstudio for the function.

Fortunately, for the LAUSD ECS logs the lines containing errors were identified during the building of the dataset. To determine which lines of code are linked in an error block, first an initial error is identified. Then the next line of code is viewed to be either an error or correct line of code. A link function was designed to automatically determine if the current line of code, i , was linked to the previous line of code, *i-1*, in the current error block. Looking at equation 4.1, we can see that link is formed by the occurrence of any one of three comparisons between the characters in lines i and *i-1*.

$$Link_{i,i-1} = \max \begin{cases} sim(function_{i,i-1}) \in (0,1) \\ sim(data_{i,i-1}) \in (0,1) \\ match(function_{i,i-1}) \in (0,1) \end{cases} \tag{4.1}$$

The first comparison checks if the functions entered in line i and *i-1* are the same or similar. Here *sim()* is a similarity function based on the Levenshtein edit distance as seen in Equation 4.2 [10]. The Levenshtein edit distance used in the similarity function is the minimum number of insertions, deletions or substitutions to transform one string into another. If a similarity of greater than the cut-off value of .75 is achieved, then the value of the comparison is set to 1 and a link is established between lines i and *i-1*. This value of .75 was tuned to fit the strings that were present in the ECS dataset [11]. This similarity function is used to allow for small spelling mistakes while still making correct links.

$$sim(function_{i,i-1}) = 1 - \frac{\text{lev}(function_{i,i-1})}{\max(function_{i,i-1})} \tag{4.2}$$

The second comparison uses this same similarity calculation, but instead compares the datasets that were used in lines i and *i-1*. Again a similarity of .75 between the named

datasets results in the value of the comparison being set to 1 and a link is established between lines i and *i-1*.

The third comparison, $match(function_{i,i-1})$, simply checks to see if the functions specified in line i are a substring of the functions specified in line *i-1*. Then the same comparison is made again to see if instead the functions specified in line *i-1* are a substring of the functions specified in line i. This comparison is made to catch errors that result from incorrect syntax or errors that come from adding extra prefixes to functions.

The automatic classification formed 623 error blocks for the inspection of errors. A manual designation of error blocks in the ECS dataset was completed and 633 error blocks were found. The incorrect results of the automatic classification resulted from 11 clusters being incorrectly linked, and one cluster being incorrectly separated. These 27 incorrectly specified clusters result in a 3.63% failure rate in the automatic clustering approach. Tuning of the cut-off value of the *sim()* function was required to reach this result. The cut-off value was decided upon to favor incorrect specifications that resulted in incorrect links instead of incorrect separations.

4.2 Evaluation of the Error Blocks

To begin our evaluation of the error blocks, we again offer an example of one of the error blocks that was formed in the ECS logs. While the error block shown in Table 4.2 is not representative of the average error block, it does show the merit of clustering errors into error blocks. Looking through the code that was entered on each attempt it is obvious that the lines of code are related. By clustering the errors into error blocks we are able to see the progress that the student made as they worked through the errors and eventually succeeded in correctly making the plot that they wanted. Without clustering these lines in some way the lines would instead be viewed as separate lines of code that were attempting a plot function. Looking at the time differences between each line we can see that it is not uniform, and that there may be value to understanding the time between related errors.

Attempt	Code	Error	Time Between
1	plot(long$lat)	Yes	0
2	plot(bus_stops$long, lat)	Yes	51.00
3	plot(bus_stops$long,lat)	Yes	53.00
4	make plot(bus_stops$long, lat)	Yes	44.00
5	MakePlot(bus_stops$long, lat)	Yes	53.00
6	Plot(bus_stops$long, lat)	Yes	8.00
7	Plot (bus_stops$long, lat)	Yes	118.00
8	plot (bus_stops$long, lat)	Yes	8.00
9	plot (bus_stops$longitude,latitude)	Yes	34.00
10	plot (bus_stops$lat, long)	Yes	19.00
11	plot(bus_stops$lat, long)	Yes	21.00
12	plot(bus_stops$lat,long)	Yes	10.00
13	plot(bus_stopslat, bus_stopslong)	No	17.00

Table 4.2: Example of a Cluster

Figure 4.1 shows that on average more than 75% of error blocks end in a correction of the original error. This is encouraging as the students are correcting their mistakes in most cases. It appears that there may be an upward trend in the average successful correction rate as the days of curriculum increase. The points on the figure represent an error block where successfully corrected blocks are the points on the top of the figure, and no corrections on the bottom. There are multiple explanations for the dip in completion at day four such as the change in data set, or the unusually low number of lines on that day that was highlighted earlier in the paper.

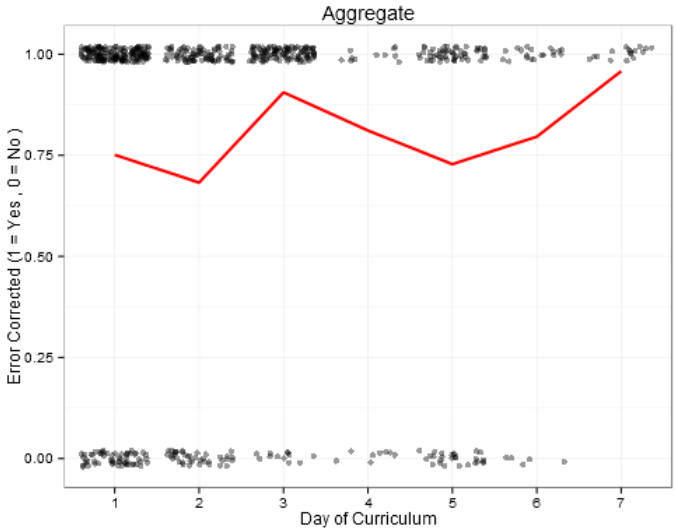

Figure 4.1: Percent of error blocks that lead to corrected error

Looking at Figure 4.2, we see that there is an increase in the average time that was spent by students on trying to correct an error. This same increase in average time is shown in Figure 4.3 when the students are not able to correct an error. This displays an increase in perseverance by spending more time to attempt at correcting an error. Additionally, we see that error blocks that had a successful correction had more time spent attempting to correct the error. This is encouraging, because it shows that students were able to persevere by spending time and completing attempts without giving up on fixing the error.

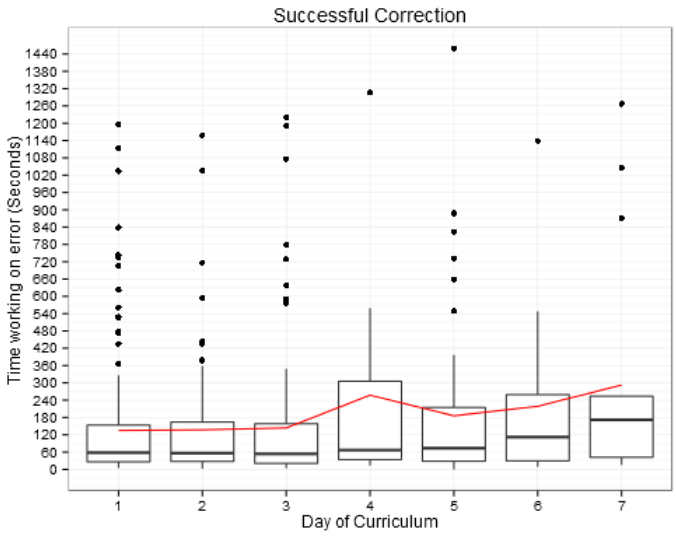

Figure 4.2: Time spent working on error block that lead to corrected error

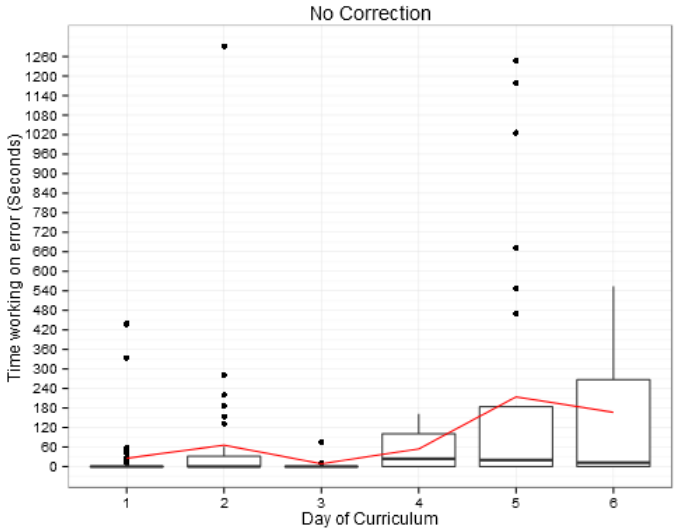

Figure 4.3: Time spent working on error block that did not lead to corrected error

The average time between each attempt at correction also shows an association with the increase in day of curriculum. Figure 4.4 & Figure 4.5 show that the students were willing to spend more time between attempts as the day of curriculum increased.

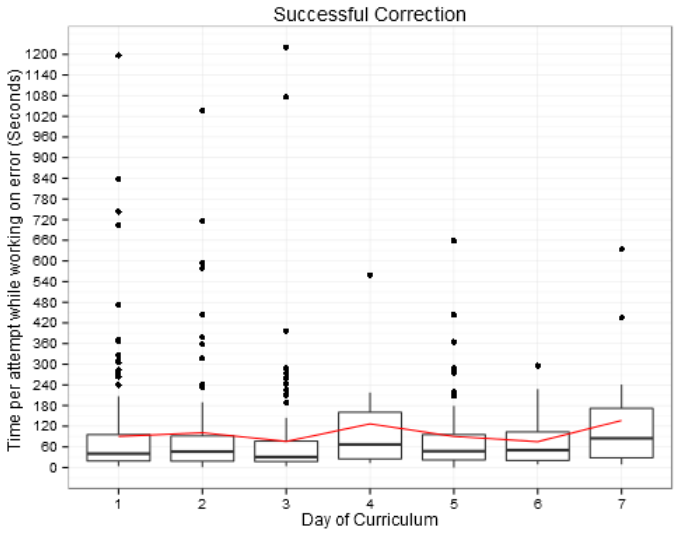

Figure 4.4: Time between attempts in an error block that lead to corrected error

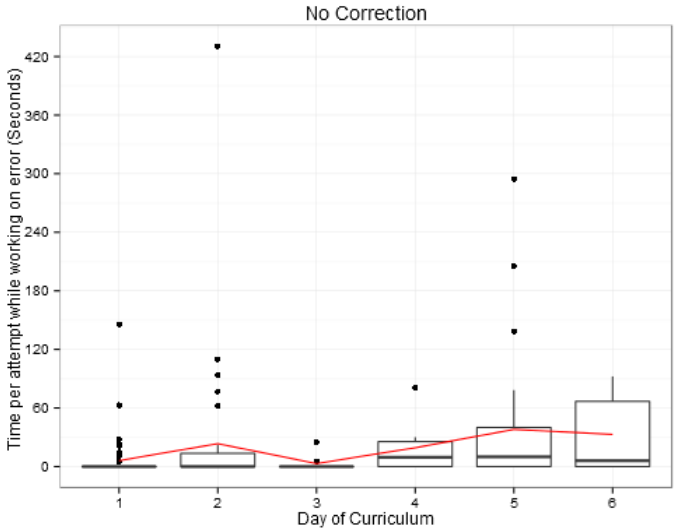

Figure 4.5: Time between attempts in an error block that did not lead to corrected error

Figure 4.6,4.7, and 4.8 highlight an encouraging fact. The majority of students tried at least one time to correct their errors. Of the errors that were never corrected, most of the students did not have a single attempted correction. Because we cannot declare directionality, this may hint at the difficulty of the initial error. Alternatively, it could suggest that encouraging students to attempt correcting errors will reduce the number of clusters that end in errors.

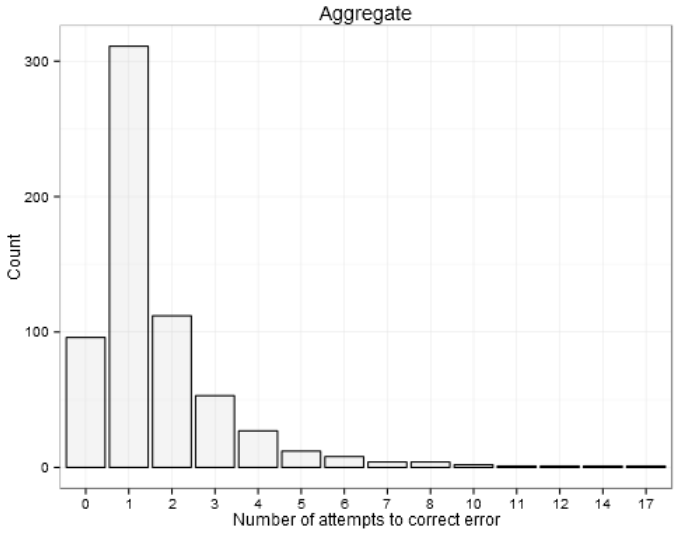

Figure 4.6: Number of attempts to correct an error

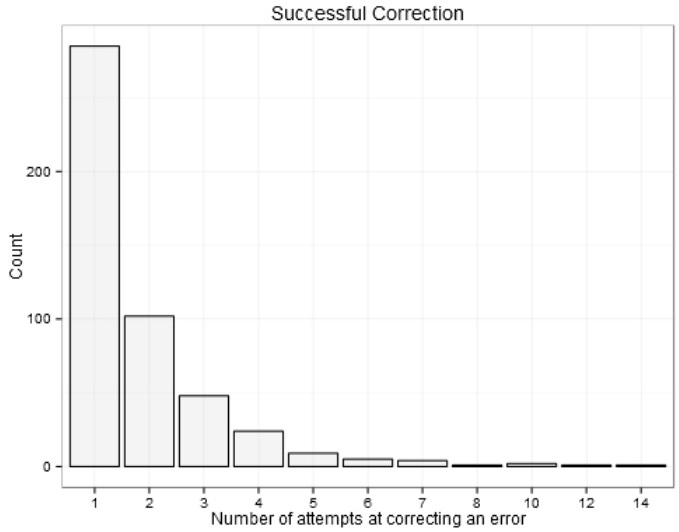

Figure 4.7: Number of attempts to correct an error - error blocks that lead to corrected error

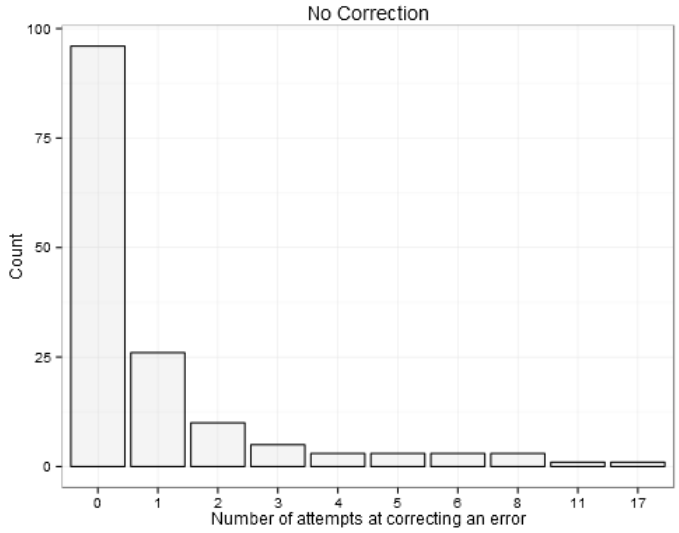

Figure 4.8: Number of attempts to correct an error - error blocks that did not lead to corrected error

Figure 4.9 shows that there is an increase in number of attempts to correct an error, as the day of curriculum increases. Again, we see that students appear to be showing more perseverance as the day of curriculum increases.

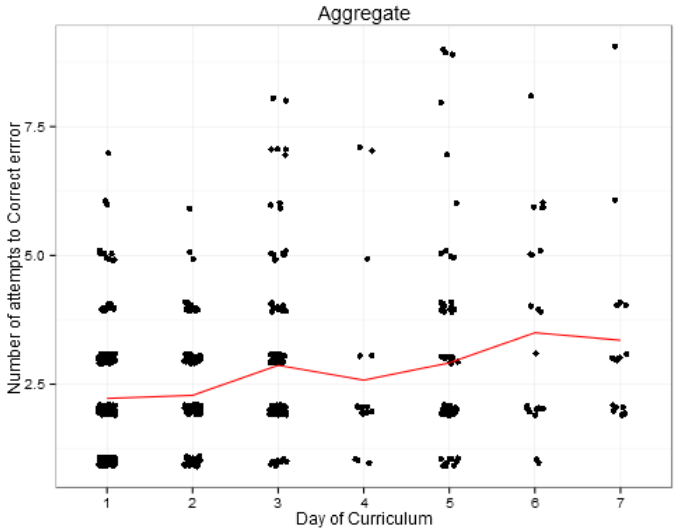

Figure 4.9: Number of attempts to correct error by day of curriculum

Earlier in the paper, Figure 3.31 showed that when each line of code was viewed independently the error rate by day showed an increase. The reason that we constructed error blocks was to provide context for each line of code, and to gather more information, so we do not have to view the lines of code independently. By rejoining the error blocks to the original data set, we can revisit this graph. Instead of viewing the error rate as a by line error rate, we can view it as a by error block error rate. For clarification, this would mean that an error block that does not end with a correction is similar to an error, and one that ends in a correction is similar to a correct line. Figure 4.10 shows that when viewed as error blocks we see a decrease in errors(error blocks that have no correction) as the day of curriculum increases. This is an important finding, because it gives value to our metric of perseverance. Instead of more errors and more time spent fixing an error meaning that the students are struggling as the day of curriculum increased, we have some justification in saying that the students are learning to fix their errors.

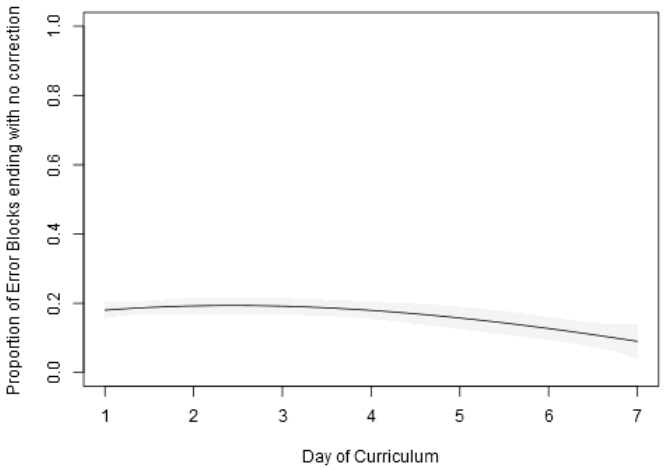

Figure 4.10: Number of attempts to correct error by day of curriculum

Eventually the knowledge learned from these clusters needs to be applied to the formative process. The student behavior when dealing with the errors can lead us to possible recommendations. If we were to recommend encouraging students to attempt to always correct their errors, what would be a good number of attempts to recommend before they are just wasting time? It appears that in Figure 4.11, four attempts at fixing an error are enough before the students experience significant drop off in success rate. This would encourage something like a "Five tries" rule before declaring themselves stuck and moving on or seeking instructor assistance.

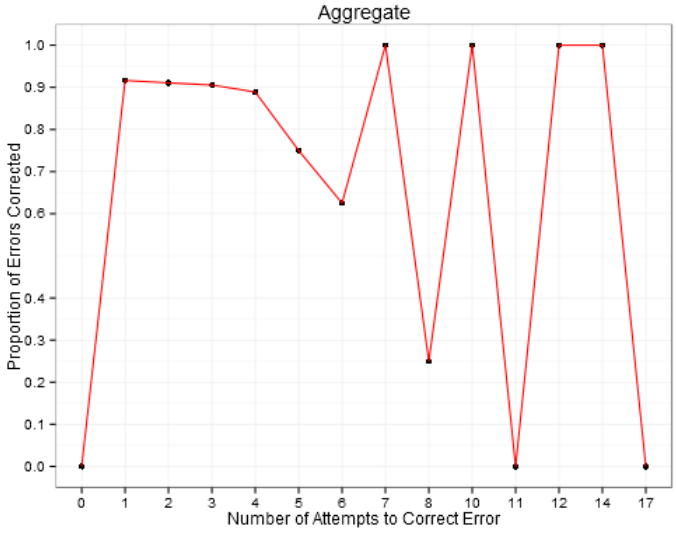

Figure 4.11: Proportion of Errors Corrected by Attempt Number

The formative assessment facilitated by the clusters does not measure learning in a summative sense. We do not receive a score for the learning a student did, but instead receive the amount of time they spent learning how to fix a code, or how long they spent trying to figure out a code before giving up. The formative assessment offers the chance to improve the syllabus with behavioral observations instead of rate the students. This behavioral framework provided by the content clusters is most importantly a valuable source of context.

CHAPTER 5

Concluding Remarks

Through multiple views we have deconstructed the student logs of the ECS introductory statistical programming course. By identifying characteristics we were able learn that there are differences between classes taught by different teachers, and even between two classes taught by the same teacher. Using time stamps we were able to discover the distribution of programming time during a class period.

Measuring learning in the classroom is a difficult task, and many metrics need to be considered when evaluating a course. Perseverance was highlighted as an important part of formative learning in this analysis. We were able to construct a content clustering approach called error blocks that provided a context to view the formative learning process. Measurement of these error blocks allowed us to assess how the students learned when trying to correct their errors. We saw that students showed increased perseverance as the days of curriculum increased by spending more time working to correct errors, and attempting more corrections. Additionally, we saw that students who successfully corrected their errors displayed increased perseverance.

Specific advice for improving the ECS curriculum, and other introductory statistical programming courses in R, can be achieved using the error blocks. For example, a rule such as "five tries" may reduce the number of errors that go without correction in a student's analysis. When the errors are viewed within the proper context the recommendations that are made are more likely to address student behaviors than when the errors are viewed as individual lines of code.

References

[1] Steven Clarke. "Evaluating a New Programming Language." *In 13th Workshop of the Psychology of Programming Interest Group*,275-289, 2001.

[2] A. Pears , S. Seidman , L. Malmi , L. Mannila , E. Adams , J. Bennedsen , M. Devlin , J. Paterson. "A survey of literature on the teaching of introductory programming, *ACM SIGCSE Bulletin*, 39(4), 2007.

[3] Rachel Cardell-Oliver. "How can software metrics help novice programmers?" *In Proceedings of the Thirteenth Australasian Computing Education Conference*, 114, 55-62, 2011.

[4] M. S. Farooq, S. A. Khan, F., Ahmad, S., Islam, A, Abid. "An Evaluation Framework And Comparative Analysis Of The Widely Used First Programming Languages." *PLoS ONE*, 9(2),2014.

[5] R Core Team, R: A Language and Environment for Statistical Computing,*R Foundation for Statistical Computing, Vienna, Austria*, 2014. http://www.R-project.org

[6] H. Wickham. ggplot2: elegant graphics for data analysis. Springer New York, 2009.

[7] Mobilize. Mobilizing for Innovative CS Teaching and Learning. Available at http://www.mobilizingcs.org/, 2010.

[8] California Common Core State Standards - Mathematics, Adopted by the California State Board of Education August 2010 and modified January 2013.

[9] V. Shute. "Focus on formative feedback." *Review of Educational Research*, 78(1), 153-189, 2008.

[10] V. I. Levenshtein. "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals". *Soviet Physics Doklady*, 10(8): 707-710, 1966.

[11] M. Cheatham, and P. Hitzler. "String Similarity Metrics For Ontology Alignment". *Lecture Notes in Computer Science*, 294-309,2013.