An Automatic Grading System for a High School-level Computational Thinking Course

Sirazum Munira Tisha
stisha1@lsu.edu
Division of Computer Science and Engineering
Louisiana State University
Baton Rouge, LA, United States

Rufino A. Oregon
rufinooregon@tamu.edu
Dept. of Education Administration and Human Resource Development
Texas A&M University
College Station, TX, United States

Gerald Baumgartner
gb@lsu.edu
Division of Computer Science and Engineering
Louisiana State University
Baton Rouge, LA, United States

Fernando Alegre
falegre@lsu.edu
Gordon A. Cain Center for STEM Literacy
Louisiana State University
Baton Rouge, LA, United States

Juana Moreno
moreno@lsu.edu
Department of Physics & Astronomy and Center for Computation & Technology
Louisiana State University
Baton Rouge, LA, United States

ABSTRACT

Automatic grading systems help lessen the load of manual grading. Most existent autograders are based on unit testing, which focuses on the correctness of the code, but has limited scope for judging code quality. Moreover, it is cumbersome to implement unit testing for evaluating graphical output code. We propose an autograder that can effectively judge the code quality of the visual output codes created by students enrolled in a high school-level computational thinking course. We aim to provide suggestions to teachers on an essential aspect of their grading, namely the level of student competency in using abstraction within their codes. A dataset from five different assignments, including open-ended problems, is used to evaluate the effectiveness of our autograder. Our initial experiments show that our method can classify the students' submissions even for open-ended problems, where existing autograders fail to do so. Additionally, survey responses from course teachers support the importance of our work.

KEYWORDS

open-ended problems, code quality, lexical analysis, machine learning

ACM Reference Format:
Sirazum Munira Tisha, Rufino A. Oregon, Gerald Baumgartner, Fernando Alegre, and Juana Moreno. 2022. An Automatic Grading System for a High School-level Computational Thinking Course . In *4th International Workshop on Software Engineering Education for the Next Generation (SEENG'22), May 17, 2022, Pittsburgh, PA, USA.* , 8 pages. https://doi.org/10.1145/3528231.3528357

SEENG'22, May 17, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9336-2/22/05…$15.00
https://doi.org/10.1145/3528231.3528357

1 INTRODUCTION AND MOTIVATION

An automatic grading system automatically assesses students' code pieces based on preset criteria and provides grades and/or feedback accordingly. Autograding systems proved very useful for courses with massive enrollments either online or in-person [20]. During the Covid-19 pandemic, while most education systems were online, instructors were stressed, exhausted, and overwhelmed with their workload [21, 27]. An automatic grading system may serve at its best by alleviating the workload for the graders. We have developed an autograder for a high-school-level course named Introduction to Computational Thinking (ICT).

As a part of the nationwide CS for All initiative [11], in 2017 the Gordon A. Cain Center at Louisiana State University launched the ICT as an introductory programming course offered to ninth and tenth graders [1]. ICT emphasizes connections to algebra, geometry, and science modeling. The ICT curriculum is designed around hands-on activities that help students build mental models of quantitative and formal reasoning and fill potential foundational gaps in their understanding. All the activities are programmed in CodeWorld [10], a web-based integrated development environment that uses a simplified version of the Haskell programming language [24]. CodeWorld uses a minimal set of graphical primitives to draw shapes, such as circles, rectangles, and text, and supports translation, rotation, scaling, and coloring of these primitives. Students are introduced to basic computational thinking practices within this environment. Figure 1 shows the CodeWorld environment displaying the code in the left panel and its outcome as a clock image in the right panel.

Since there are not enough computer science teachers, the ICT curriculum was designed to be accessible to teachers with diverse backgrounds. Prospective ICT teachers receive intensive training in the summer before teaching the course. The ICT curriculum includes about a hundred coding assignments. It is undoubtedly a hectic and time-consuming task for a novice teacher to grade all the coding assignments manually. Additionally, in this course, all the assignments have graphical output. To grade this type of

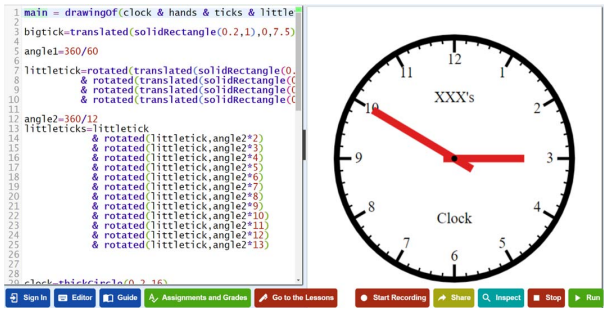


Figure 1: CodeWorld environment with editor panel on the left and the respective code output picture on the right.

assignments by only examining the final output is not sufficient to provide meaningful feedback to the students; checking the quality or elegance of the code is also crucial. Here in this research, the term "code elegance" includes the code's efficiency, coherency, and readability aspects, such as appropriate variable naming, spacing, function usage, code refactoring, documentation etc. Measuring code elegance is essential in assessing students' computational thinking abilities. Our target is to develop an autograder that can evaluate the code elegance of programming assignments even before running them and provide crucial insights into the student learning of basic computational thinking skills.

Most available autograder systems, such as Gradescope [16], LAB.COMPUTER [22], and CodeGrade [9], use the unit testing approach where the graders create several input test cases. The code is run for all the test cases; if all the outputs are as expected, the code is considered correct. This approach works best for programs where outputs are literal. However, it cannot assess pictorial or graphical outcomes. On top of this, it cannot evaluate codes for open-ended assignments where the students' creativity is encouraged. The unit testing approach is also inconvenient for novice teachers who barely have any computer science or coding background. To reduce the test case writing loads, Carnegie Mellon University's CS1 course [26] provides an autograder with built-in test cases. However, the built-in test cases confine students to the exact reference code restricting their creativity. As the ICT course encourages students to demonstrate their creativity, an autograder with built-in test cases is not a good fit for the curriculum.

The goal of our novel autograder is not to replace teachers by strictly grading the programming assignments but rather to reduce the grading efforts by helping teachers assess the different aspects of a piece of code.

2 RELATED WORK

We aim to provide feedback on the writing styles of the pieces of code without execution. Static program analysis analyzes computer software without executing any program. Schwieren et al. [30] and Benac et al. [5] used static analysis in their proposed autograder. In their review, Arifi et al. [2] described the necessity of using both dynamic and static analysis, and they found relatively good results using the fusing system. Edwards et al. [13] in Web-CAT autograder also used a combination of static analysis and unit testing. In most cases, the static analysis is performed on some version of a program's source code and, in other cases, on some form of its object code. A static analysis tool or linter, namely HLint [19] in our case, can suggest alternative functions that might be more efficient or simplify a code and reduce redundancy. Still, it lacks the rubrics that a high-school teacher might want to look for in this introductory level course, such as a specific function that the problem assignment asked to use, or redundant use of a function to draw something to achieve the learning objectives. Therefore, in our case, a simple lexical analysis is not sufficient to measure code elegance.

Researchers further implemented machine learning approaches to build an autograder. For instance, Srikant et al. [32] implemented Regression and SVM (Support Vector Machine), using features such as keywords from codes, control flow graphs, and data dependency graphs. In later work [33], they chose features which were the most represented in the sample programs, or the most correlated features. Barstad et al. [4] implemented K-nearest-neighbors (KNN), Naive Bayes (NB), and Decision trees (DTree) for classifying their data into well-written and poorly written categories using features from static analysis. Similarly, Singh et al. [31] extracted standard features like operators, expressions, etc. Gupta et al. [17] converted pieces of code to sequences of tokens using Pygments, a Python syntax highlighter, and used a deep neural network to analyze and classify relevant and non-relevant code. In our approach, we also used Pygments. Recently, Wu et al. [34] implemented personalized feedback for the online CS1 level course named "Code in place" [29] using machine learning approaches. To get the features, they used the built-in parser to tokenize the input, using features such as compilation error, function and method names. They used CodeBERT [14] to train their dataset. In our work we implement machine learning approaches considering several facts described in the latter part of this paper.

3 DATASET EXPLORATION

For developing our autograder, we evaluate the submissions associated with five different assignments from different units of the ICT course curriculum. We manually grade the assignments by analyzing the source codes and categorize them based on the approach taken by the student. We observe how multiple aspects of the code vary when different approaches are used to solve the same problem. The detailed procedure followed is described next.

3.1 Classifying Assignments

We start our discussion with the clock assignment. This problem requires drawing an analog clock including hour and minute hands, hour numbers, hour ticks, and minute ticks. A typical output is shown in the right panel of Fig. 1. A sample code is shown in Listing 1. We select this assignment because it is assigned halfway through the course curriculum when students are already familiar with basic coding. We extract all the submissions for this problem from the ICT database, and filter out the submissions that were not Haskell code. We keep all other submissions regardless of whether they were incomplete, incorrect, or did not compile or run, as we are only judging the source code elegance. Since, the grades provided by the teachers were mainly based on the correctness and creativity

of the drawn output, to conduct this study, we manually grade the assignments as either elegant or non-elegant.

We observe students write their clock drawing codes in four different ways shown in Figure 2. Most of the students took the naive approach of drawing all sixty-minute ticks. Others tried to optimize their code by drawing a quadrant of the minute ticks and rotate it three times, drawing twelve-minute ticks and rotate the block four times, or drawing only the four-minute ticks between two-hour ticks and rotate them twelve times. Since for this particular problem, students were encouraged to start thinking about optimizing the code, we consider repetitive code as non-elegant, and the codes where abstractions were used as elegant codes.

We also observe that most optimized codes have fewer lines than non-optimized ones. So we set a threshold for the number of lines in the code to check whether it was sufficient to measure elegance. However, we observe that the number of lines of code is not enough to detect non-elegant codes where the students plugged direct calculations into the draw function without assigning the calculated values to any variable. This approach ends up with fewer lines of code. The opposite happens for detecting an elegant code that uses more variables. So the classification of the codes based only on the number of lines of code is not sufficient for measuring the sophistication (elegance) of the code. At a minimum, we need to incorporate the number of variables, operators, and functions used in the code.

```
import Standard
import Extras.Colors
import Extras.Cw(overlays)

program = drawingOf(pivotPoint
                    & bigHand
                    & smallHand
                    & overlays(hourNumber,12)
                    & overlays(hourTickMarks,12)
                    & overlays(minuteTickMarks,60)
                    & myName
                    & clockLettering
                    & innerCircle
                    & circleOutline
                    & background
                    )


circleOutline = solidCircle(9)
innerCircle = painted(solidCircle(8.5), "maroon")
pivotPoint = solidCircle(0.25)
bigHand = translated(painted(rotated(solidRectangle
    (0.5,9),-60), "red"),3,1.75)
smallHand = translated(painted(rotated(solidRectangle
    (0.5,6),60), "red"),-1.5,0.9)

myName = painted(translated(lettering(""), 0,4), "yellow"
    )
clockLettering = painted(translated(lettering("Clock"),
    0,-4), "yellow")

background = painted(solidRectangle(20,20), "lightblue")

hourTickMarks(hourTickNumber) = rotated(translated(
    solidRectangle(0.5,1.5), 0,8), hourTickNumber * 30)

minuteTickMarks(minuteTickNumber) = rotated(translated(
    solidRectangle(0.2,0.75), 0,8.25), minuteTickNumber
    * 6)

hourNumber(clockNumber) = rotated(translated(rotated(
    dilated(lettering(printed(clockNumber)),1.5),
    clockNumber*30),0,6.3),clockNumber*(-30))
```

Listing 1: Clock sample code

```
Drawing of minute ticks
BEGIN PROGRAM
        Draw (minTick1)
        Draw (minTick2)
        Draw (minTick3)
        .....
        Draw (minTick60)
END
```

(a) Sixty minute ticks are drawn individually (Non-elegant)

```
Drawing of minute ticks
BEGIN PROGRAM
        Draw(Quadrant of minute ticks )
                Draw (minTick1)
                Draw (minTick2)
                ....
                Draw (minTick15)
        End
        Draw Quadrant
        Draw(Rotate(Quadrant),angle1)
        Draw(Rotate(Quadrant),angle2)
        Draw(Rotate(Quadrant),angle3)
END
```

(b) Draw a quadrant of minute ticks and rotate them three times (Elegant)

```
Drawing of minute ticks
BEGIN PROGRAM
        Draw(minTick set : 12 minute ticks , 30 degree apart)
                Draw (minTick1)
                Draw (minTick2)
                ....
                Draw (minTick12)
        End
        Draw (minTick set )
        Draw(Rotate(minTick set ),angle1)
        Draw(Rotate(minTick set ),angle2)
        Draw(Rotate(minTick set ),angle3)
        Draw(Rotate(minTick set ),angle4)
END
```

(c) Draw twelve minute ticks and rotate them four times (Elegant)

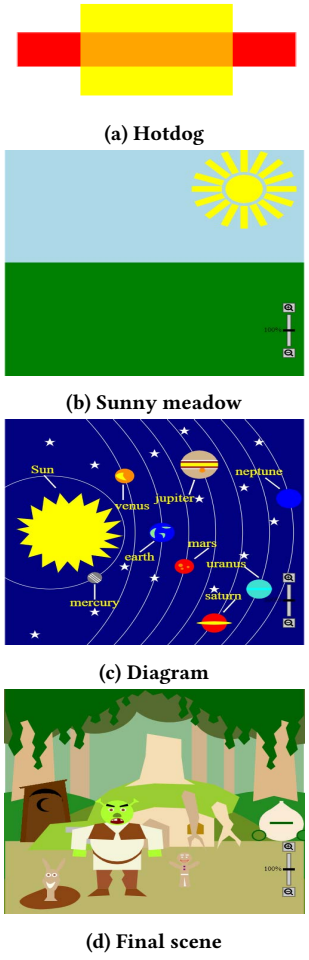
```
Drawing of minute ticks
BEGIN PROGRAM
        Draw(minTick set : 4-minute ticks within two-hour ticks)
                Draw (minTick1)
                Draw (minTick2)
                ....
                Draw (minTick4)
        End
        Draw (minTick set )
        Draw(Rotate(minTick set ),angle1)
        Draw(Rotate(minTick set ),angle2)
        Draw(Rotate(minTick set ),angle3)
        ....
        Draw(Rotate(minTick set ),angle12)
END
```

(d) Draw only the four minute ticks between two hour ticks and rotate them twelve times (Elegant)

Figure 2: Four types of codes found in student submissions. (a) Drawing sixty minute ticks separately is considered naive and non-elegant. (b)-(d) The concept of looping is encouraged and graded as elegant code.

To refine our approach and choose appropriate parameters to characterize the code, we analyze additional activities of different nature. We chose the very basic hot dog drawing (Fig. 3(a)), and the sunny meadow drawing (Fig. 3(b)) where students were asked to produce similar output. In addition to those, we chose two arbitrary drawing activities where students were asked to draw any diagram they had studied in their science courses (Fig. 3(c)), and the final group project of drawing a creative scene (Fig. 3(d)). For data collection, we again only filter out the blank submissions and non-Haskell codes and keep the rest of the data. While manually grading these assignments, we observe that the number of lines of code could be an essential aspect for grading the elegance of the hotdog and the sunny meadow assignments, but not for the arbitrary diagram and the scene as students were encouraged to show their creativity in both activities. We further observe that for the free drawing problems it was difficult to follow the code and understand it if proper spacing, comments, definitions, and indentation were not used. In other words, readability is the main element of elegance for these open-ended problems. In brief, code efficiency, coherency, and consistency are the aspects of a source code that measure code elegance for diverse assignments.

Now similarly as the line of code measurement, it is not suitable to judge the elegance of a piece of code according to a preset threshold for other code features like number of spaces, comments usage or variable naming. For instance, a non-elegant code may have more spacing that makes it disorganized, or may have many variable names and comments that are irrelevant. Considering these situations an autograder based on threshold judgements might not be able to address the elegance of the code. It would also be quite tedious to adjust the thresholds for different problems. Machine learning approaches may be useful in this respect. We believe that if we train our system with a manually graded training set by expert graders, then the system will be able to judge the assignments accordingly.



(a) Hotdog

(b) Sunny meadow

(c) Diagram

(d) Final scene

Figure 3: Sample outputs of (a) hotdog, (b) sunny meadow, (c) diagram, and (d) final scene assignments.

3.2 Research Questions

To extract the aspects of code elegance observed in our dataset, we perform a lexical analysis of the student-submitted code. Our proposition is that machine learning classifiers might infer the human grading approaches from the distribution of the lexer features in the source code. In addition, we believe that an autograder based on assignments graded by experts can be beneficial for novice teachers to gain competency. We presume that a machine learning approach trained on an expert evaluated dataset can predict and suggest better rubrics on new datasets.

Therefore, as an initial attempt, we use a lexer to extract the features from the student codes and fed these features to different machine learning classifiers. To guide our research, we formulate the following research questions.

- **RQ1:** How can a simple lexer identify the important features that represent the coding aspects?
- **RQ2:** Can machine learning algorithms successfully categorize aspects of the code?
- **RQ3:** How do different models compare?

4 IMPLEMENTATION

4.1 Ground Truth Preparation

We start our initial experiment by manually classifying the samples in Elegant vs. Non-elegant. Two graders manually classified the assignments (a graduate student and an undergrad student). Later, to train our machine learning classifiers we used a balanced dataset of 300 samples of the clock, 722 samples of hot dogs, 454 samples of sunny meadow, 684 samples of the diagram, and 608 samples of the scene activity. While manually classifying the activities as elegant and non-elegant, we consider the following:

(1) Is the algorithm used efficiently?
(2) Is the code easy to read and follow (proper usage of spaces, comments, and meaningful variable names)?

For the specific problems where students have to produce the same output type, i.e., clock, hot dog, and sunny meadow, we consider the efficient use of algorithms as the main characteristic of an elegant code. On the other hand, we chose readability and coherency as the main ingredient of elegant codes for

open-ended problems, i.e., diagram and final scene projects. We also look for proper spacing and comments usage for classifying these last two assignments. Datasets are available at https://git.brbytes.org/tisha/Autograder_dataset.

4.2 Experimental Analysis

We presume that the frequency of repeating the same element in the code is related to the student level of algorithmic understanding. By analyzing and classifying these features in clusters we will help teachers to better judge the quality of their student submissions. Therefore we clustered our features. However, we also surmise that these features would not be enough to judge the correctness and creativity of the assignments.

4.3 Qualitative Analysis

We use the following pipeline in our qualitative analysis:

(1) Collect samples from the database,
(2) Extract features from the samples,
(3) Cluster the features,
(4) Fit the models with previously clustered features,
(5) Compare the results and choose the best model.

We extract 14 problem-independent features from the students' codes using a simple Python lexer, Pygments, a syntax highlighter [7] able to identify Haskell keywords. We further count the Haskell keywords according to their category. We calculate some Halstead complexity measures, [18] such as volume of the code, difficulty to write or understand the program, and effort to coding, from the lexer outcome. We also check if the variable names are meaningful and count them. All the features we consider are:

(1) Lines of code
(2) Operator usage
(3) Comments usage
(4) Number of blank lines/ white space
(5) Number of local variables
(6) Number of non-local variables
(7) Integer count
(8) Float count
(9) String count
(10) Halstead volume
(11) Halstead difficulty
(12) Halstead effort
(13) Number of meaningful variables
(14) Number of meaningless variables

Along with these features, we have our manual classification of Elegant vs. Non-elegant as a target function. We check the distribution of these features in students' codes for separate assignments to categorize them accordingly. Next, we classify students' submissions according to their code aspects and suggest to teachers which class the submission fall into rather than providing direct scores. We use KNN+ cluster analysis [3][1]. We automatically extract the cluster boundaries by using the elbow method [25]. Fig. 4 shows a histogram displaying the number of lines of code for the clock problem after automatic clustering.

[1]We also used expectation-maximization clustering, but did not get good results.

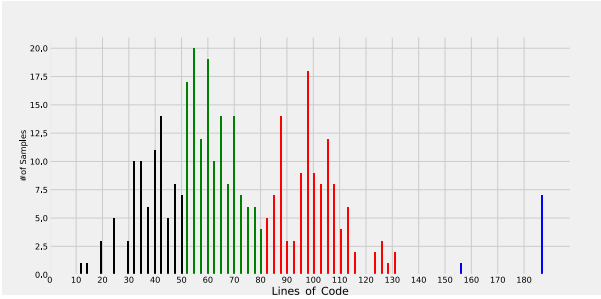


Figure 4: Sample of automatic clustering for the number of lines of code in the clock activity. The number of clusters is four for this feature and activity.

We then feed all these feature classes and our manual grading to machine learning classification models using a pipeline written in Python and the Scikit-learn library [28]. Classification algorithms like Naive Bayes (NB), KNN (K-nearest Neighbor), Support Vector Machine (SVM) and Decision Tree (DTree) were previously used for text and code analysis [4, 15, 32, 33]. Motivated by these prior studies, we experiment with these four machine learning models. While training our classifiers, we used 10-fold cross validation to avoid over-fitting. Later, to find out the most crucial features for the models, we use a wrapper sub-set evaluation. Table 1 shows up to four best features from the wrapper sub-set evaluation based on the accuracy values of the models.

We also calculate precision and recall to evaluate the performance of different classifiers. Precision is the ratio of correctly classified instances of a specific label to the total number of cases classified under that label. On the other hand, recall represents the ratio between correctly classified cases for a label and the number of cases that belong to that label. We collect precision and recall values for each activity and machine learning model with clustered features and with clustered and wrapper-subset features. The average precision and recall values are shown in Table 2.

5 RESULTS AND DISCUSSION

While false-positive situations where a non-elegant student code is identified as elegant might be acceptable, false-negative instances where competent students may be penalized even though their submissions are elegant must be reduced. We also aim to develop an accurate automatic grading system with reduced false values. Therefore, we intend to use a machine learning model that produces acceptable precision and recall measures.

Table 2 shows the recorded precision (Pre) and recall (Rec) values. We observe that clustered features performed similarly for almost all the models except NB. We observe in the histograms of multiple features (Figure 4) that most features do not follow a normal distribution. Naive Bayes (NB) performs better for normally distributed datasets as it is a generative models. On the other hand, SVM, KNN, and decision tree-based models are discriminative models and we observe they performed similarly [6]. We further notice that clustered+wrapper features produce better results. The recall values are between 71% and 95% and precision values are in the 73-93%

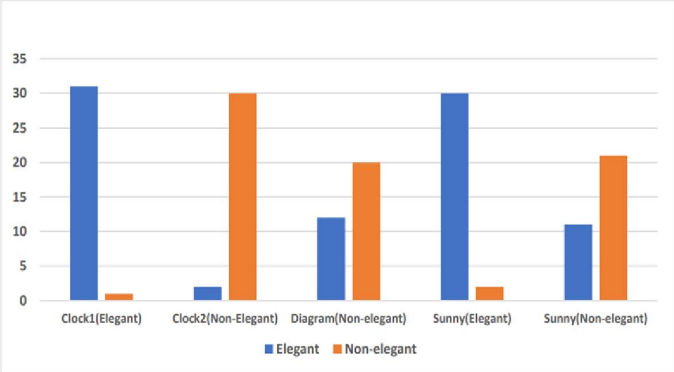


Figure 5: Survey response with teachers grading input. The horizontal axis displays the problem name with the autograder suggestions in parenthesis. The vertical axis shows teachers scoring as elegant and non-elegant.

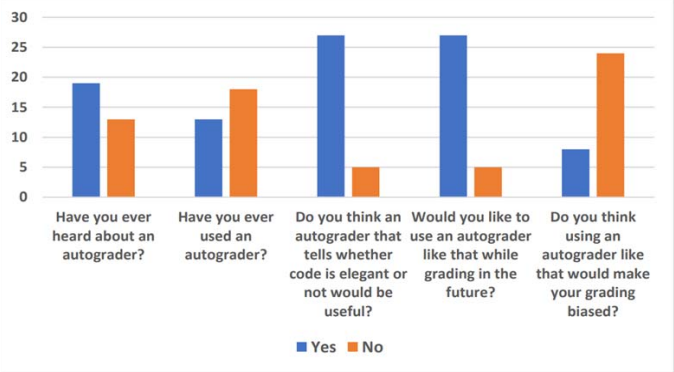


Figure 6: Responses to yes/no survey questions

range. Overall the decision tree algorithm with clustered+wrapper features produced slightly better precision and recall results.

In brief, to answer our research questions, we can conclude from these observations that a simple lexer counting code features in combination with machine learning models can successfully classify elegant vs non-elegant pieces of code. Our results also show that the decision tree with wrapper subset features performs slightly better than order machine learning models.

6 SURVEY

To check the applicability and acceptance of our approach we conducted an initial survey in which 32 ICT course teachers from various disciplines (Computer Science, Math, Arts, Spanish, Chemistry etc.) and from several local high schools participated. The survey is available at https://lsu.formstack.com/forms/elegant_autograder. We provided five sample student codes, autograder suggestions based on the decision tree classifier, and possible influential factors in the scoring of that particular piece of code. Teachers evaluated two clock examples, one diagram example, and two sunny meadow examples. We asked the participants whether they agreed or disagreed with the autograder choice. Later we asked three additional yes/no answer questions, which are:

(1) Do you think an autograder that tells whether code is elegant or not would be useful?
(2) Would you like to use an autograder like that while grading in the future?
(3) Do you think using an autograder like that would make your grading biased?

Figure 5 shows the scoring responses. From the teacher responses, we find that for the *clock1* sample which autograder suggested it was elegant, 31 teachers agreed with the autograder classification, and one disagreed. Similarly, for the *clock2* selection, 30 teachers agreed, and two dissented, for the *diagram*, 20 teachers agreed and 12 disagreed, for the *sunny1* sample, 30 teachers agreed, and two disagreed, and finally for *sunny2*, 21 teachers agreed, and 11 disagreed.

In relation with the responses to our yes/no questions, among 32 teachers, twenty-seven teachers would like to use the autograder in the future, and 24 teachers thought the autograder would not make their grading biased. In addition, 27 teachers thought the autograder suggestions were helpful. Figure 6 shows the chart for these responses.

We also asked participants to provide their comments or suggestions to improve the autograder. Most of the teachers suggested having more scales than only elegant and non-elegant. They also suggested to have a measurement of effort. We find that the novice teachers felt more comfortable and relieved about using the autograder feedback than the expert teachers. Our target was to help novice teachers mainly; this revealed our success in reaching our goal. Furthermore, teachers thought overall, the autograder's suggestion would be helpful to point out different aspects and start grading from there.

7 FUTURE WORK

Our initial results show the applicability of a machine learning classifier based on lexer features to automatically judge code elegance. In this research we extract features from Haskell pieces of code and classify the submissions in binary classes, i.e., elegant and non-elegant. We are hopeful that a similar approach will work to judge code elegance for other programming languages. We extract 14 features and implement four machine learning classifiers with promising results. We are planning to extract additional features like separated operator count, style of variable names to measure the style consistency, use of magic numbers, etc. and implement other classifiers along with modern boosting algorithms like the XGBoost [8] and CatBoost classifiers [12]. Additionally, to check the applicability of our approach to predict correctness we experiment with the clock problem only. All the machine learning classifiers produce around 50% of incorrectly classified instances, confirming our intuition that our approach with features extracted by lexer is not suitable for judging correctness. Judging creativity depends upon the graders' perspectives on the output picture, which is not possible to measure with our approach either. Our understanding is that along with lexer, if we extract features from the structure of the code, code matrices, and features from both compile-time and run-time information such as semantic analysis [23], parse tree, and

Table 1: Best lexer features according to wrapper subset evaluation technique based on accuracy values

Problem set	NB subset	SVM subset	KNN subset	DTree subset
Clock	Lines of code Comments usage No of blank lines	Lines of code Comments usage Halstead effort	Lines of code Float counts No of meaningless variables	Lines of code Halstead effort No of meaningless variables
Hot-dog	String count Halstead volume	No of local variables Integer count String count	Operator usage String count Halstead volume	No of local variable String count Halstead volume
Sunny meadow	Integer count Float count	No of local variables No of non-local variables No of meaningful variables No of meaningless variables	Operator usage No of local variables No of non-local variables Halstead effort	Lines of code Halstead difficulty No of meaningful variables No of meaningless variables
Diagram	Comments usage No of blank lines Integer count	Comments usage No of blank lines String count	Comments usage No of blank lines Float count	Comments usage No of blank lines No of meaningful variables
Scene	Lines of code Comments usage No of blank lines	Operator usage Comments usage No of meaningful variables	Operator usage Comments usage No of meaningful variables	Comments usage No of blank lines No of meaningful variables

Table 2: Comparison of classifiers for all activities with clustered features and with only the sub-set of features from the wrapper subset evaluation. Recall values are in bold. Results show overall the decision tree with wrapper subset features perform better.

Classifiers	Feature Choice	Clock		Hotdog		Sunny		Diagram		Scene	
		Pre	Rec	Pre	Rec	Pre	Rec	Pre	Rec	Pre	Rec
NB	Clustered	0.90	**0.62**	0.92	**0.75**	0.86	**0.94**	0.65	**0.63**	0.58	**0.74**
	Clustered+Wrapper	0.83	**0.88**	0.88	**0.95**	0.87	**0.94**	0.70	**0.66**	0.61	**0.76**
SVM	Clustered	0.84	**0.88**	0.89	**0.94**	0.88	**0.94**	0.76	**0.63**	0.76	**0.71**
	Clustered+Wrapper	0.85	**0.89**	0.91	**0.93**	0.90	**0.94**	0.75	**0.70**	0.73	**0.70**
KNN	Clustered	0.84	**0.87**	0.93	**0.87**	0.87	**0.88**	0.72	**0.60**	0.72	**0.73**
	Clustered+Wrapper	0.86	**0.86**	0.89	**0.91**	0.90	**0.89**	0.67	**0.71**	0.73	**0.63**
DTree	Clustered	0.80	**0.82**	0.91	**0.88**	0.90	**0.82**	0.75	**0.76**	0.81	**0.75**
	Clustered+Wrapper	0.89	**0.86**	0.93	**0.94**	0.92	**0.93**	0.74	**0.71**	0.73	**0.75**

call-graphs, scene-graphs, etc., we will be able to judge the code structure, code complexity, and correctness. Our future research target is to include these features. Additionally, we would also like to address plagiarism testing. Our intuition is that by comparing the lexer-extracted features, plagiarism issues could be addressed.

Moreover, in this initial implementation we manually graded the samples by briefly looking at the code. In the future we would like to follow a well developed rubric while preparing the ground truth. We also have plans to classify our data in more categories not only the binary classes elegant vs. non-elegant. We would also like to use grades provided by the teachers as ground truth to make our model more robust. Our future goal is to have a large-scale study and deploy student surveys to better understand which kind of feedback they value more.

The long-term goal of our research work is to develop a fully functional, automated, and robust grading system for the ICT course specifically, and later for other coding courses to support teachers better with grading suggestions, and to alleviate their manual grading load.

8 ACKNOWLEDGMENT

This research is funded by the NSF Computer Science for All: Researcher Practitioner Partnership program (CNS-1923573) and the U.S. Department of Education, Education Innovation and Research program (U411C190287). RAO is funded by NSF award OAC-1852454.

9 AUTHOR'S PROFILE

Sirazum Munira Tisha is a Ph.D. candidate in the Division of Computer Science and Engineering at Louisiana State University (LSU). Currently, she is developing an automated grading system for the Introduction to Computational Thinking (ICT) course. She served

as course instructor at the LSU STEM Pathways Summer Teacher Institute program. Her research interest includes Automated Grading Systems, Computer Science Education and Curriculum Development, and Machine Learning.

Rufino A. Oregon is a senior undergraduate student majoring in Technology Management in the college of Educational Administration & Human Resource Development at Texas A&M University (TAMU). He worked as a teacher assistant for seven semesters where he taught programming and engineering in freshmen engineering courses at TAMU. Rufino is interested in inter-disciplinary areas that intersect with computer science and has participated in computational biology and engineering enculturation research projects.

Gerald Baumgartner received a Dipl.-Ing. degree from the University of Linz, Austria, and M.S. and Ph.D. degrees from Purdue University, all in computer science. He began his academic career at The Ohio State University in 1997. Since 2004 he is in the Division of Computer Science & Engineering at Louisiana State University. His research interest includes cloud computing middleware, compiler optimizations, the design and implementation of domain-specific and object-oriented languages, and development and testing tools.

Fernando Alegre is associate director of the Gordon A. Cain Center, LSU. He received his M.S. in Computer Science from the Georgia Institute of Technology. He has experience working in research-oriented projects and implementing scientific software in fields such as image and signal processing, statistics, Monte Carlo methods, machine learning, Bayesian inference, probabilistic graphical models, compiler writing, automatic program analysis and abstract interpretation.

Juana Moreno is a professor in the LSU Department of Physics & Astronomy with a joint appointment in the LSU Center for Computation & Technology. She received her Ph.D. in Physics from Rutgers University. She also serves as the team leader of the BRBytes project which is part of the CSforALL movement. Her research interests include computing education and computational condensed matter physics.

REFERENCES

[1] Fernando Alegre, John Underwoood, Juana Moreno, and Mario Alegre. 2020. Introduction to Computational Thinking: A New High School Curriculum Using CodeWorld. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 992–998. https://doi.org/10.1145/3328778.3366960

[2] Sara Mernissi Arifi, Ismail Nait Abdellah, Azeddine Zahi, and Rachid Benabbou. 2015. Automatic program assessment using static and dynamic analysis. In *2015 Third World Conference on Complex Systems (WCCS)*. 1–6. https://doi.org/10.1109/ICoCS.2015.7483289

[3] David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding*. Technical Report. Stanford.

[4] Vera Barstad, Morten Goodwin, and Terje Gjøsæter. 2014. Predicting source code quality with static analysis and machine learning. In *Norsk IKT-konferanse for forskning og utdanning*.

[5] Clara Benac Earle, Lars-Åke Fredlund, and John Hughes. 2016. Automatic Grading of Programming Exercises Using Property-Based Testing. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 47–52. https://doi.org/10.1145/2899415.2899443

[6] Guillaume Bouchard and Bill Triggs. 2004. The tradeoff between generative and discriminative classifiers. In *16th IASC International Symposium on Computational Statistics (COMPSTAT'04)*. 721–728.

[7] Georg Brandl, Matthäus Chajdas, Armin Ronacher, Pocoo, and Tim Hatch. 2006. *Python syntax highlighter*. https://pygments.org

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[9] CodeGrade. 2020. *Deliver engaging feedback on code.* https://www.codegrade.com

[10] CodeWorld. 2020. *Educational computer programming environment using Haskell.* https://github.com/google/codeworld

[11] CSforALL. 2021. CSforAll. https://www.csforall.org/.

[12] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. 2018. CatBoost: gradient boosting with categorical features support. *ArXiv* abs/1810.11363 (2018).

[13] Stephen H Edwards and Manuel A Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*. 328–328.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[15] Chase Geigle, ChengXiang Zhai, and Duncan C Ferguson. 2016. An exploration of automated grading of complex assignments. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. 351–360.

[16] Gradescope. 2020. *Online grading platform.* https://www.gradescope.com

[17] Anshul Gupta and Neel Sundaresan. 2018. Intelligent code reviews using deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*.

[18] MH Halstead and T McCabe. 1976. A software complexity measure. *IEEE Trans. Software Engineering* 2, 12 (1976), 308–320.

[19] HLint. 2020. *Linter for Haskell.* https://github.com/ndmitchell/hlint

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

[21] Florian Klapproth, Lisa Federkeil, Franziska Heinschke, and Tanja Jungmann. 2020. Teachers' Experiences of Stress and Their Coping Strategies during COVID-19 Induced Distance Teaching. *Journal of Pedagogical Research* 4, 4 (2020), 444–452.

[22] LAB.COMPUTER. 2020. *Put your computer laboratory on the browser.* https://lab.computer

[23] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic Grading of Programming Assignments: A Formal Semantics Based Approach. In *Proceedings 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training ICSE-SEET 2019, 25-31 May 2019, Montreal, Canada*.

[24] Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)* (2010).

[25] Dhendra Marutho, Sunarna Hendra Handaka, Ekaprana Wijaya, et al. 2018. The determination of cluster number at k-mean using elbow method and purity evaluation on headline news. In *2018 International Seminar on Application for Technology of Information and Communication*. IEEE, 533–538.

[26] Carnegie Mellon University's School of Computer Science (SCS). 2020. *Carnegie Mellon University Computer Science Academy.* https://academy.cs.cmu.edu/

[27] Gabriella Oliveira, Jorge Grenha Teixeira, Ana Torres, and Carla Morais. 2021. An exploratory study on the emergency remote education experience of higher education students and teachers during the COVID-19 pandemic. *British Journal of Educational Technology* (2021).

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[29] Christopher Piech, Ali Malik, Kylie Jue, and Mehran Sahami. 2021. Code in Place: Online Section Leading for Scalable Human-Centered Learning. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 973–979.

[30] Joachim Schwieren, Gottfried Vossen, and Peter Westerkamp. 2006. Using Software Testing Techniques for Efficient Handling of Programming Exercises in an e-Learning Platform. *Electronic Journal of e-Learning* 4, 1 (2006), 87–94.

[31] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. 2016. Question independent grading using machine learning: The case of computer program grading. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 263–272.

[32] Shashank Srikant and Varun Aggarwal. 2013. Automatic grading of computer programs: A machine learning approach. In *2013 12th International Conference on Machine Learning and Applications*, Vol. 1. IEEE, 85–92.

[33] Shashank Srikant and Varun Aggarwal. 2014. A System to Grade Computer Programming Skills Using Machine Learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 1887–1896. https://doi.org/10.1145/2623330.2623377

[34] Mike Wu, Noah Goodman, Chris Piech, and Chelsea Finn. 2021. ProtoTransformer: A Meta-Learning Approach to Providing Student Feedback. *arXiv preprint arXiv:2107.14035* (2021).