Real-time Metacognition Feedback for Introductory Programming Using Machine Learning

Phyllis J. Beck
Computer Science and Software Engineering
Mississippi State University
Mississippi State, MS, USA
pjb82@msstate.edu

M. Jean Mohammadi-Aragh
Electrical and Computer Engineering
Mississippi State University
Mississippi State, MS, USA
jean@ece.msstate.edu

Christopher Archibald
Computer Science and Software Engineering
Mississippi State University
Mississippi State, MS, USA
archibald@cse.msstate.edu

Bryan A. Jones
Electrical and Computer Engineering
Mississippi State University
Mississippi State, MS, USA
bjones@ece.msstate.edu

Amy Barton
Technical Communication Program
Mississippi State University
Mississippi State, MS, USA
abarton@engr.msstate.edu

***Abstract*— This is a Work in Progress Research to Practice Category paper. Research has shown that novice programmers struggle with learning introductory concepts and find it difficult to monitor their own progress. Teachers often have hundreds of students and multiple sections of programming courses to teach, making it infeasible to provide the amount of independent feedback each student may need to flourish. With limited instructor feedback, students who can self-monitor and self-assess their programming metacognition have a higher chance of developing a process for solving programming challenges. In this paper, we expand on the literate programming paradigm by using natural language processing and machine learning methods to automatically analyze and classify student programming metacognition levels through their source code comments. Our intent is to ultimately integrate our classification models into an interactive developer environment to provide real-time feedback to students about their metacognition while learning to program.**

Keywords— pedagogy; novice programmers; learning analytics; research-to-practice

I. Introduction

A continuing challenge in computer science education is providing enough independent feedback to students as they learn to program. Research has shown that novice programmers struggle with learning introductory concepts and find it difficult to monitor their own progress [1]. Increasing enrollment at many campuses has resulted in larger rosters and more frequent sections of programming courses. As the student-to-teacher ratio increases, educators find themselves with even less time per student and are unable to provide individual feedback needed for concept mastery. With limited instructor feedback, students must rely on their own abilities to recognize and correct programming deficiencies in order to develop a process for solving programming challenges. Successful self-monitoring by novice programmers is possible, but not probable, and likely one reason novices struggle to learn to program.

As part of a larger project investigating the use of Writing to Learn (WTL) activities to promote student programming metacognition, we propose providing automated, individual feedback based on student thinking processes and visual organization that are visible in student source code comments. We hypothesize that immediate automated feedback can both relieve the workload of professors and guide students towards increasing their strategic knowledge by developing a more effective problem solving process.

A. Writing to Learn to Program

WTL is based on the premise that students learn through the act of writing [2-3]. While students often view writing as separate from the practical demands of their professions, WTL strategies, when integrated purposefully, blend writing and learning in a way that has been shown to support student development in multiple disciplines. WTL deepens student understanding, improves student engagement, increases retention, and makes students active participants in the learning process [4-5].

Based on WTL successes in other disciplines, our overarching research project is investigating WTL in the context of programming. We believe that through writing, programmers can organize their learning, allowing them to view the coding process as a series of logical, interrelated steps that connect an overall goal to all the potential ways of reaching it. WTL strategies such as intermingled commentary and reflection while coding should enable students to move beyond surface understanding to analysis and synthesis of difficult processes [6]. Knuth, the creator of the literate programming paradigm certainly believed that writing was essential for programmers: "Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do." [7]

In our investigation of WTL to support learning to program, we have incorporated WTL strategies into introductory computer programming laboratory assignments and are comparing student work from those laboratories with student work from traditional laboratories. In order to minimize

This material is based upon work supported by the National Science Foundation under Grant No. DUE-1612132. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

additional work and time requirements for the WTL students, our initial investigation has focused on examining existing writing in the form of source code comments. We used coding experts to classify student source comments. Our initial results indicate that source code comments can indeed convey information about student thinking processes and their organizational patterns [8]. Our other research is currently linking source code comment classifications to student performance. One practical implication of our research is that we can provide feedback to students to encourage them to use thinking processes and organizational styles that are correlated with stronger student performance. For example, if a student exclusively employs literal source code comments, we could encourage them to use conceptual or reflective comments. In order to fully harness this opportunity for providing feedback, we are investigating methods to automatically classify source code comments in real-time. This work-in-progress paper focuses on that effort.

B. Feedback Promotes Learning

Within the context of instructional design, feedback includes both formative and summative evaluation results provided to students in order to promote learning. Formative feedback is continuous feedback throughout the learning module (e.g., feedback on draft essays). Summative feedback is feedback given at the end of a learning module (e.g., feedback through a chapter test, grade on a final program submission). Many classrooms ignore formative feedback, focus on summative feedback, and move on to new material. However, feedback is most valuable when it is given during the learning process because it provides students a chance to identify and correct misunderstandings or improve learning deficiencies [9].

Within the context of writing and composition, feedback is considered a critical element of the learning process. Best practices of writing instruction compel instructors to provide a recursive revision process before the final paper is submitted for review [10-11]. Formative feedback on writing is valuable for improving student learning whether it comes from an instructor or peers [12]. Considering source code comments as a form of writing compels educators to consider ways to provide formative feedback during the coding process.

II. Predicting Students' Visual Organization Strategy

The first phase of implementing real-time feedback for students in an introductory programming course is developing the classification models that will be integrated into the students' IDE to promote formative feedback. In this paper, we focus on the feature engineering process and model for classifying a student's visual organizational strategy. The current dataset contains 98 completed lab assignments where each sample is an executable python file from a pair programming student lab assignment, which we will refer to as a "program". Each program was anonymized by removing all header information within the file and given a classification according to our current codebook [8]. We developed the data set for training our machine learning classifiers using a supervised learning approach where each program was hand classified by a domain expert.

The qualitative classifications that can be assigned to a program for students' visual organization are Block-level, Unitization, Every-line, Insufficient, or None. A *Block-level* classification demonstrates program code and comments that are characterized by blocks of comments followed by large blocks of code. *Unitization* is characterized by code and comments that are grouped into logical sections where the unit contains comments and code that are highly related to a particular function or action being executed in the source code. An *Every-line* strategy is characterised by a pattern of a single comments that precede or follow a line or lines of code. Every-line units are small, and there tends to be an excessive amount of commenting throughout the source code. A classification of *Insufficient* occurs where there are too few comments in the source code to determine a meaningful organizational strategy; code is often disorganized and commenting, if present, is sporadic. *None* is a default classification that occurs if the are no comments in the source code. For a detailed explanation of these classifications please refer to [8].

III. Feature Engineering

Most traditional machine learning approaches require a numerical representation of the input data. This numerical representation consists of a vector of separate features generated from each example. In this section, we explore the challenge of generating features from student programs that allow for accurate classification of the student's visual organizational strategy. For classification, features need to represent the structure and patterns of data in a constant-sized feature vector. This is, the set of features generated from each data sample needs to be the same size. Source code presents challenges as each program varies in length.

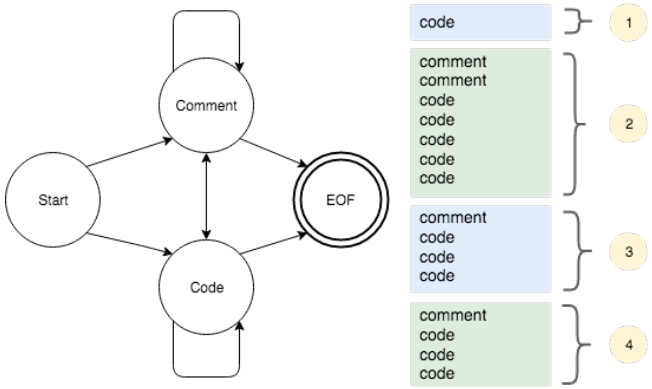
The classification of the student programs into the different visual organization strategies primarily depends on the number and arrangement of code and comment lines. Motivated by this observation, our initial focus was to derive features from an abstraction of each sample program. The abstraction replaced each line of text with a single label, either 'comment' or 'code' while ignoring white space. Inline comments are considered code lines. This is because the typical inline comment is very short, often one to three words and would be considered an insignificant comment according to our Thinking Processes classification system [8]. (We note that inline comments were rare in our dataset and appeared in only one of our samples. In the future, we may investigate other methods of handling inline comments.) An example abstraction of sample program WL5_S5_G5 is shown in Fig. 1.

```
['code','comment','comment','code','code','code',
   'code','code','comment,'code','code','code',
  'comment','code','code','code','comment','code',
'comment','code','comment','code','comment','code',
     'comment','code','comment','code','code']
```

Fig. 1. Example feature vector for the executable sample WL5_S5_G5

All our initial features focused on capturing the relationships between the code and comment lines of the

program using our abstraction. Conceptually, we break down a program into two groupings. First, a *section* consists of consecutive lines with the same classification (e.g., "comment"). The next grouping is a *unit*, which consists of a comment section followed by a code section. (Note that the initial unit may not have an initial comment section.) The features we investigate can be generated from the properties of the sections and units in a given program. To compute the features, we utilize a simple state machine as illustrated in Fig. 2a. This state machine keeps track of the current section type (code/comment), how many lines are in each section, when the section type changes, how many lines are in the sections that make up a unit, and how many units there are in the program. A *state change* occurs when a section changes from one type to another. We reach the terminal state when we reach the end of the file. The state machine produces all of the values necessary to calculate the features that we describe. An illustration of how the first 16 lines of the WL5_S5_G5 sample breaks down into sections and units can be seen in Fig. 2b. We can see that these lines break down into seven sections (code section, comment section, code section, comment section, code section, comment section, and code section), which make up four individual units (1, 2, 3, and 4), and result in six state transitions (code to comment, comment to code, code to comment, comment to code, code to comment, and comment to code).



(a) (b)

Fig. 2. (a) State machine to compute features from program abstraction. (b) Example feature vector for the executable sample WL5_S5_G5

Table 1 describes the 16 features we investigated. For completeness, we included some features even though we anticipated they would not be useful. For example, we did not expect the number of lines in a file to be useful since it varies with the complexity of the lab assignment. However, since it was a simple to generate feature, we evaluated it.

IV. Results

To analyze the performance our initial feature set, which uses multiple categories, we used Multinomial Logistic Regression [13], which generalizes logistic regression for a multiclass classification problem. The original dataset only contained a single Block-level sample and a few None samples, so to reduce the complexity and to remove issues associated with having an unbalanced number of samples for each class, we removed these samples from the dataset and focused on classifying three categories: Unitization, Every-line and Insufficient. With Logistic Regression, we can predict the probability that a program belongs to a given class and choose the predicted class based on the highest probability.

TABLE I. Feature Descriptions

Feature	Description
line_total	Number of total lines in the program abstraction, includes all code and comments.
num_of_state_changes	Number of times the state changes from code to comment or from comment to code.
statechanges_vs_linetotal	Number of state changes divided by the total number of lines in a program. This feature is meant to capture how frequent the state changes are.
units_vs_linetotal	Number of units divided by the total number of lines in a program.
total_num_of_comments	Number of comments in a program.
total_num_of_code	Number of code lines in a program.
ratio_comment_to_code	Number of comments divided by the number of code lines in a program.
num_of_units	Number of units in a program.
avg_unit_ratio	A unit ratio is the number of comment lines in that unit divided by the number of code lines in that unit. This feature is the average unit ratio over all units in the program
avg_length_of_units	The average number of lines in each unit in the program.
max_comments	Maximum length of a comment section.
max_code	Maximum length of a code section.
avg_comments	Average length of a comment section.
avg_code	Average length of a code section
avg_unitlen_vs_ratio	The average length of a unit divided by the avg_unit_ratio for that program.
avgunit_vs_numunit	The average length of a unit divided by the total number of units in an executable.

To evaluate the quality and usefulness of each feature, a logistic regression classifier was trained using 5-fold cross-validation with a single feature as the input. This was repeated for each feature in our feature set. We utilized scikit-learn's grid search to determine the best parameters for the classifier [14]. To measure how well each classifier performed, we examined the null accuracy and the model score. The null accuracy refers to how accurate the classifier would be if we predicted every sample as the most frequent class in the dataset. The model score is the percentage of accurate predictions achieved on the test set. We split our 98 samples and used 81 as the training set and 17 as the test set. The most frequent class was *Unitization* (8 out of 17 test samples). Therefore, the null accuracy was 47 percent (if we classified all

samples as Unitization we would achieve an accuracy of 47 percent). Table 2 lists the model score for the multinomial logistic regression classifier trained with each feature. The best classification accuracy achieved was 88 percent. All but two features performed better than the null accuracy. Features such as line_total, total_num_of_code, avg_comments, max_comments performed poorly on their own. The most useful features were ratio_comment_to_code and avg_unitlen_vs_ratio, which performed as well as the Top 15, Top 10 and Top 5 feature combinations.

TABLE II. LOGISTIC REGRESSION CLASSIFIER RESULTS

Features	Model Score	Precision
Top 15	0.8823	0.90
Top 10	0.8823	0.89
Top 5	0.8823	0.89
avg_unitlen_vs_ratio	0.8823	0.89
ratio_comment_to_code	0.8823	0.91
All Features	0.8235	0.82
avgunit_vs_numunit	0.8235	0.84
avg_code	0.8235	0.82
num_of_units	0.8235	0.84
num_of_state_changes	0.8235	0.84
statechanges_vs_linetotal	0.7647	0.76
units_vs_linetotal	0.7647	0.76
total_num_of_comments	0.7647	0.79
avg_unit_ratio	0.7647	0.67
avg_length_of_units	0.7647	0.76
max_code	0.6470	0.66
line_total	0.5882	0.57
total_num_of_code	0.5294	0.59
max_comments	0.4705	0.22
avg_comments	0.4705	0.22

To better understand how the best features appear in data, a table of these features, along with the mean and standard deviation of that feature on all examples from each class, is shown in Table 3. This illustrates how well features are able to differentiate the classes. In general, the Every-line and Insufficient categories appear to be the most distinct. The Insufficient category demonstrates a high degree of variability, which is to be expected but also introduces difficulties when attempting to classify an observation based on demonstrated patterns. Every-line is characterized by a large number of units that are small with a large comment to code ratio and a very small avgunit_vs_numunit ratio of 0.19 that has a very low standard deviation of 0.06. Unitization has fewer units and more comments on average with a smaller comment to code ratio. The avgunit_vs_numunit is on average 0.48 with a standard deviation of 0.23. Insufficient is characterized by high ratios in avgunit_vs_numunit with an average of 2.72 and a standard deviation of 4.67. It is interesting to note that features such as line_total, total_num_of_code, total_num_comments do not contribute much information on their own due to the variation in lab assignments. If this was a single assignment, these features would carry more weight, but instead there is a need to look to composite features such avg unit ratio versus the average unit length, which provide a more generalized feature for classification across multiple lab assignments

V. CONCLUSION

This work in progress paper presents our initial efforts to engineer a feature set from student programming assignments to describe the patterns students use to visually organize their code. We used this feature set to train a multinomial logistic regression classifier and were able to achieve a mode score of 88%. This positive result from our initial investigation supports our efforts of developing a real-time feedback system for introductory programming courses. Future research includes developing additional features for improving the accuracy of the model, expanding the dataset to include more Block-level and None samples, integrating the ‘inline’ classification into the vector representation of a program, and testing additional classifiers and machine learning methods. Additionally, we plan to integrate all the classification models into an IDE to provide real-time feedback to students on their learning process and programming metacognition.

TABLE III. LOGISTIC REGRESSION CLASSIFIER RESULTS

Feature	Statistic	Uniti-zation	Every-line	In-sufficient
avg_length_of_units	mean	4.187152	2.741714	7.258412
	std	1.208258	0.331523	3.982370
avg_unitlen_vs_ratio	mean	8.360554	3.170356	49.940628
	std	6.009565	0.635563	93.543432
avgunit_vs_numunit	mean	0.484181	0.197639	2.727165
	std	0.232034	0.063301	4.671904
num_of_units	mean	9.608696	15.171429	5.058824
	std	2.653955	4.680911	2.331056
ratio_comment_to_code	mean	0.421408	0.693579	0.268110
	std	0.179754	0.181290	0.222321
total_num_of_code	mean	28.173913	24.314286	23.647059
	std	12.805041	6.452411	3.773943
total_num_of_comments	mean	10.956522	16.714286	6.117647
	std	4.361226	5.953602	4.498366

REFERENCES

[1] Lahtinen, E., Ala-Mutka, K. & Järvinen, H. M. (2005). A study of the difficulties of novice programmers . ACM SIGCSE Bulletin, 37(3), 14-18.

[2] Russell, D.R. (1991). *Writing in the academic disciplines, 1870–1990: A curricular history*. Carbondale, IL: Southern Illinois UP.

[3] J Linton, P., Madigan, R., & Johnson, S. Introducing students to disciplinary genres: The role of the general composition course. (1994). *Language and Learning across the Disciplines*, 1(2), 62–78.

[4] Paretti, M.C., Theories of Language and Content Together: The Case for Interdisciplinarity. Across the Disciplines, 2011. 8(3).

[5] Paretti, M.C., When the Teacher is the Audience: Assignment Design and Assessment in the Absence of "Real" Readers, in Engaging Audience: Writing in an Age of New Literacies, A. Gonzalez, E. Weiser, and B. Fehler, Editors. 2009, NCTE Press: Urbana, IL. p. 165-185.

[6] Jones, B.A., Mohammadi-Aragh, M.J., Barton, A.K., Reese, D., & Pan, H. (2015). Writing-to-Learn-to-Program: Examining the need for a new genre in programming pedagogy. *122nd ASEE Annual Conference and Exposition*, Seattle, Washington.

[7] Knuth, D. "Literate Programming (1984)" in Literate Programming. CSLI, 1992, pg. 99.

[8] Mohammadi-Aragh, M.J., Beck, P.J., Barton, A.K., Reese, D., Jones, B.A., Jankun-Kelly, M. (2018). Coding the coders: A qualitative investigation of students' commenting patterns. *American Society for Engineering Education Annual Conference and Exposition*, Salt Lake City, UT.

[9] Bransford, J. D., Brown, A. L., and Cocking, R. R. (Eds.) (2000). *How people learn: Brain, mind, experience, and school*. Washington, D.C.: National Academy Press.

[10] Hayes, J. R., Flower, L., Schriver, K. A., Stratman, J., & Carey, L. "Cognitive processes in revision," Reading, writing, and language processing. Advances in applied psycholinguistics. S. Rosenberg (Ed.), Vol. 2. Cambridge, UK: Cambridge University Press. 1987. 176-240.

[11] Sommers, N. Revision strategies of student writers and experienced writers. College Composition and Communication, 31, 1980. 378-387.

[12] Cho, K. and MacArthur, C. "Student revision with peer and expert reviewing." *Learning and Instruction*. 20. 2010. 328-338.

[13] James, G., Witten, D., Hastie, T., & Tibshirani, R. *An introduction to statistical learning*. 2013 New York: springer.

[14] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O. & Vanderplas, J. "Scikit-learn: Machine learning in Python." *Journal of machine learning research*, 12, 2011, 2825-2830.