2013 12th International Conference on Machine Learning and Applications

Automatic Grading of Computer Programs : A Machine Learning Approach

Shashank Srikant
Aspiring Minds
Email: shashank.srikant@aspiringminds.in

Varun Aggarwal
Aspiring Minds
Email: varun@aspiringminds.in

***Abstract*—The automatic evaluation of computer programs is a nascent area of research with a potential for large-scale impact. Extant program assessment systems score mostly based on the number of test-cases passed, providing no insight into the competency of the programmer. In this paper, we present a machine learning framework to automatically grade computer programs. We propose a set of highly-informative features, derived from the abstract representations of a given program, that capture the program's functionality. These features are then used to learn a model to grade the programs, which are built against evaluations done by experts on the basis of a rubric. We show that regression modeling based on the given features provide much better grading than the ubiquitous test-case-pass based grading and rivals the grading accuracy of other open-response problems such as essay grading . We also show that our novel features add significant value over and above basic keyword/expression count features. In addition to this, we propose a novel way of posing computer-program grading as a one-class modeling problem. Our preliminary investigations in the same show promising results and suggest an implicit correlation of our features with the proposed grading-levels (rubric). To the best of the authors' knowledge, this is the first time machine learning has been applied to the problem of grading programs. The work is timely with regard to the recent boom in Massively Online Open Courseware (MOOCs), which promises to produce a significant amount of hand-graded digitized data.**

I. Introduction

Automatic evaluation of human-skills is a topic of keen interest today. Specific to this is the evaluation and assessment of a person's ability to write computer programs. The automatic grading of computer programs has immense value not only in the recruitment processes of software development companies but also in teaching programming to students in training institutes, universities and Massively Open Online Courses (MOOCs). The benefits of such automatic systems are aplenty - firstly, it reduces the time and effort of graders - the number of problems given out to a candidate and the number of candidates being assessed need not be limited by the availability of graders. Secondly, it can provide real-time feedback. Candidates would not have to wait for the availability of teaching assistants or a faculty member to learn about the quality of their program to improve themselves. Currently, the scalability of MOOCs teaching programming and algorithms is impeded by this constraint, which such an automatic system would help resolve. Thirdly, it has the potential to lead to some standardization in the assessment of computer programs. Assessments today vary from grader to grader, with no underlying framework for reference.

A primary aspect of such an automated grading system has to be the ability to grade a computer program on the basis of a rubric. Such a rubric would typically map a score (quantitative measure) to the ability of the programmer to solve the problem. For instance, a high-ranging score could mean that the candidate's program is a "Correct and efficient implementation of the problem", whereas a lower grade could mean that the candidate's program "Shows emergence of some of the needed control structures and data operations, but requires improvement". Such a mapping to a rubric is essential in order to provide an objective feedback, thereby helping candidates write better solutions.

The most widespread approach currently used for automatic assessment of programs is by evaluating the number of test-cases they pass[1][2][3][4]. Unfortunately, this approach is wrought with problems. Programs which pass a high number of test-cases may not be efficient and may have been written with bad programming practices. On the other hand, programs that pass a low number of test-cases are many-a-times quite close to the correct solution; some unforced or inadvertent errors make them eventually fail the suite of test-cases designed for the problem[1]. Lastly, a score which is quantitatively defined as the number of test-cases-passed completely disregards the requirement of the score to map to a human-intuitive rubric of program quality.

Another popular approach to the automated grading of programs makes use of measuring the similarity between abstract representations (such as Control Flow Graphs and Program Dependence Graphs) of a candidate's program and representations of correct implementations for the problem[5][6][7][8][9]. Although promising, the theoretical elegance is damaged by the existence of multiple abstract representations for a correct solution to a given problem. Secondly, there is no underlying rubric that guides the similarity metric and neither are approaches to map the metric to a rubric discussed.[2] Apart from this, there have been publications on automatic correction of small programs [10] and peer-based assessments of programs [11], neither of which directly addresses the problem of automatic grading of programs.

We present a new approach to automatically grade computer programs (referred henceforth as just *programs* in the discussions) using machine learning. In our approach, we derive highly-informative features that capture the program's functionality from abstract representations of a given program.

[1] In our own work, we see 40-50% programs falling in this category. Refer to Section IV for details.

[2] *Janicic et al.* [8] use regression to do so, but on a very coarse set of features without much room to match the evaluation.

978-0-7695-5144-9/13 $31.00 © 2013 IEEE
DOI 10.1109/ICMLA.2013.22

These features are then used to learn a model to grade the programs against evaluations done by experts on the basis of a rubric. To the best of the authors' knowledge, this is the first time machine learning has been applied to the grading of programs.

Specifically, the paper makes the following contributions:

- We introduce a novel grammar of features which captures signature elements in a program that human experts recognize when assigning a grade. In essence, these features capture the functionality of the program. We show these features to correlate well with a proposed problem-independent rubric.
- We show empirically that the novel features add value by better modeling human-grading than an elementary keyword-counts model. (see Section II for details.)
- We introduce a framework with a preliminary demonstration of how learning can be used to solve the problem when one is constrained by having an unbalanced or a small number of graded programs.

The paper is organized as follows: Section II introduces a rubric to grade programs and proposes a grammar to generate informative features from a program. In Section III, we discuss a machine learning framework to solve the problem. In Section IV, we present the experiment details and results. Finally, Section V discusses the results and future work.

II. FEATURES FOR PROGRAM GRADING

A. Decoding the human evaluation process

Our primary motivation to address the problem of automatic evaluation was to try and understand the aspects which a manual evaluator would consider when grading a program. We understand that for a given problem, there are certain signature features which the evaluator looks for. These features are in terms of specific control structures, keywords or data dependencies being present in the program. As an illustration, we analyze a program written for a problem requiring to print a pattern (see Table I for problem statement and the corresponding program). The sample program is a pseudo-code in which the declaration statements of variables have been omitted and a generic syntax to print a variable has been used. This program is not fully correct as it misses out on printing a newline character at the beginning of every iteration of the outer-loop.

Given an integer ***N***, write a program to print ***N*** lines of the following pattern of numbers

```
1
2  3
3  4  5
4  5  6  7
...
```

```
void print_lines(int N){
    for(i = 1; i <=N; i++){
        count = i;
        for(j = 0; j < i; j++){
            print(count);
            count++;
        }
    }
}
```

TABLE I: Pattern-printing problem with a sample program

A human evaluator going through the sample program mentioned in Table I would consider the following signature features -

– Looks for basic keywords - Is a variable's value being printed? Are there loops? Are variables being incremented? If they exist, it demonstrates that the candidate has at least some basic idea of the constructs needed for the problem at hand.

– Is there a nested loop? Is a print statement included in the nested loop? If they exist, it can be inferred that the candidate has realized that a 2-dimensional printing operation primarily requires a nested loop to access each unit along the two axes.

– Is the terminating condition of the outer loop based on the input to the function? Is the terminating condition of the inner loop dependent on a variable defined in the outer loop? If yes, the candidate understands that the specific pattern in the problem demands some relationship between the two axes to exist, translating to data-dependencies between the two loops.

– Is the argument to the print function in the inner loop dependent on a variable which changes its values in each iteration? Specifically, does it increment by one and get updated to a value which is incremented by one in each outer loop? Is there a print statement in the outer loop to control the newline characters in the pattern? If yes, the candidate has a very clear understanding of the nuances demanded by the problem.

TABLE II: Rubric to grade computer programs

SCORE	INTERPRETATION
5	**Completely correct and efficient:** An efficient implementation of the problem using the right control structures, data-dependencies and consideration of nuances/corner conditions of the logic.
4	**Correct with silly errors:** Correct control structures and critical data-dependencies incorporated. Some silly mistakes fail the code to pass test-cases.
3	**Inconsistent logical structures:** Right control structures exist with few/partially correct data dependencies.
2	**Emerging basic structures:** Appropriate keywords and tokens present, showing some understanding of a part of the problem.
1	**Gibberish Code** Seemingly unrelated to the problem at hand.

It is evident that with the appearance of each set of signature features, in the order described above, a human evaluator would provide a higher score - the absence of the right keywords would receive the lowest score; the presence of the right keywords would be awarded some points; the presence of a nested loop structure would be awarded some more; the presence of data-dependencies between the loops would be awarded a relatively high score, with the best score being awarded to those programs which have all the problem-specific nuances such as the right re-initialization in the outer loops and a newline character being printed in the right loop, in addition to having all other features mentioned thus far.

This simple approach helps motivate both - the design of a grading rubric and the identification of the right features in a program which would predict it's correctness.

At this point, we urge the reader to exercise caution and not be misled by the apparent simplicity of the illustration described above. The implementation discussed above may be written with numerous variations - *while* loops replacing

the *for* loops, varying number of variables and expressions, varying data-structures (use of 2-D arrays) and even varying algorithms (recursion to iterate). The question of grading the program remains hard and such a simplified illustration acts as a primer to understand the nuances involved in the process.

B. Evaluation Rubric

Guided by the observations mentioned in the previous Section, we present in this Section a problem-independent rubric to grade a program solving a programming problem. For any program, we define the state it is in by considering how much it has advanced in solving the given problem. The semantics used in the program i.e. the kind of keywords, expressions, control structures and data-dependencies specific to the problem at hand help decide how well developed a solution it is.

The rubric is presented in Table II. It objectively captures the program's state which, we hypothesize, links to the candidate's ability to develop an *algorithm* for a given problem. It can be used across problems and leaves scope to be more detailed and fine-grained.

C. Grammar of Features

In this Section, we describe a grammar of features which helps in establishing a congruence between a program and the functionality it performs. Motivated by the rubric's design, the features capture various states of expressions, control structures, data-dependencies and other properties that exist in a program. We present the features as belonging to broad classes, where each class categorises them by the underlying property they capture. Since the states mentioned in the rubric attempt to capture how well developed an *algorithm* is, these features are applicable to a program written in any programming language. Moreover, being generic, they are invariant of the underlying problem and to any particular implementation of a given problem. The details of these classes are as follows -

a. Basic Features: This class includes features obtained by counting the occurrences of various keywords and tokens appearing in the source code. These include keywords related to control structures such as for, $while$, $break$, etc., operators defined by a language like '+','-','$*$', '%', etc., character constants used in the program like '0', '1', '2', '100', etc., external function calls made like $print()$, $count()$, etc.

These counts are useful to see whether the right constructs even appear in a program (characterstics of Score 2, Table II) irrespective of whether the control-flow and data-dependencies are right.

b. Expression Features: This class includes features obtained by counting the occurrences of expressions appearing in a program. An expression in a programming language is a combination of explicit values, constants, variables, operators and functions. The expressions used in a program help identify arithmetic and relational operations typical of the underlying algorithm.

These expressions can be described with a varying degree of detail. For instance, the counts can either be characterized by the operators used, or by the external functions called, or by the constants used (such as '+', '-', '$print$', '100' etc.). These could further be classified by counting specific instances of each such operator, function-call etc. or by counting the total operators, function-calls etc. that appear in an expression. In our work, we capture operator-specific and data-type-specific information to count the different *types* of expressions used in a program. Table III illustrates a few examples of expression-features and provides their significance.

TABLE III: Sample features and their interpretation

Feature	Interpretation
LP_LP {*Basic*}	The number of times a nested loop (a for-in-a-for or a $while$-in-a-for etc.) appears in the program
%::*LP_IF* {*Basic c.c** }	The number of times a modulus operator (%) appears in a nested conditional-in-a-loop. The operator appearing in a statement like for{.. $if(x$%2==0)..} would be captured by this feature
v:1::*op*:%::*c*:'2' {*Expression*}	The number of times an expression containing a modulus operator (%), one variable and the constant 2 appears in the program. An expression like x%2==0 would be captured by this feature
v:2::*op*:!= ::*IF* {*Expression c.c**}	The number of times an expression containing a not-equal-to operator (!=), two variables and no constants appears in an if block in the program. An expression like ..$if(x$!=$y)$.. would be captured by this feature
v:2::*op*:< ↑ v:1::*op*:++ {*Data-dep*}	The number of times an expression containing a relational operator (<) and two variables is dependent on an expression containing a post-increment operator and one variable
v:2::*op*:<::*LP_LP* ↑ v:1::*op*:++::*LP* {*Data-dep c.c**}	The number of times an expression containing a relational operator (<) and two variables, which appear in a nested loop, is dependent on an expression containing a post-increment operator and one variable appearing in a loop

* c.c : in control-context

c. Basic & Expression Features in Control Context: This class includes features obtained by counting the occurrences of various expressions, keywords, tokens etc. (described in (*a*) and (*b*) above) in *context* of the control-flow structures they appear within. A *context* here signifies a block-scope extended by a control-structure to which a section of the program belongs to. A control-flow structure could include loop-statements (for, $while$, do-$while$, etc.) or conditional-statements (if, if-$else$, $switch$, etc.). In essence, these features give us the control structures the features described in (*a*) and (*b*) belong to. As in the earlier class of features, the control context too can be characterized by varying levels of details - counts could be specific to instances of loops and conditionals like for, if or could be generic to the occurrence of loops irrespective of whether it is a for or a $while$. Counts could also vary by the depth of the nested structures, where only the most recent context could be counted as against counting the exact context the expressions/keywords appear in. In our work, we do not differentiate between different instances of loops and conditionals and work with exact depths of contexts.

For the sample problem on printing a pattern, the CFG (see Fig. 1) reveals the presence of two occurrences of the post-increment operator (in $count$++ and j++) appearing in a

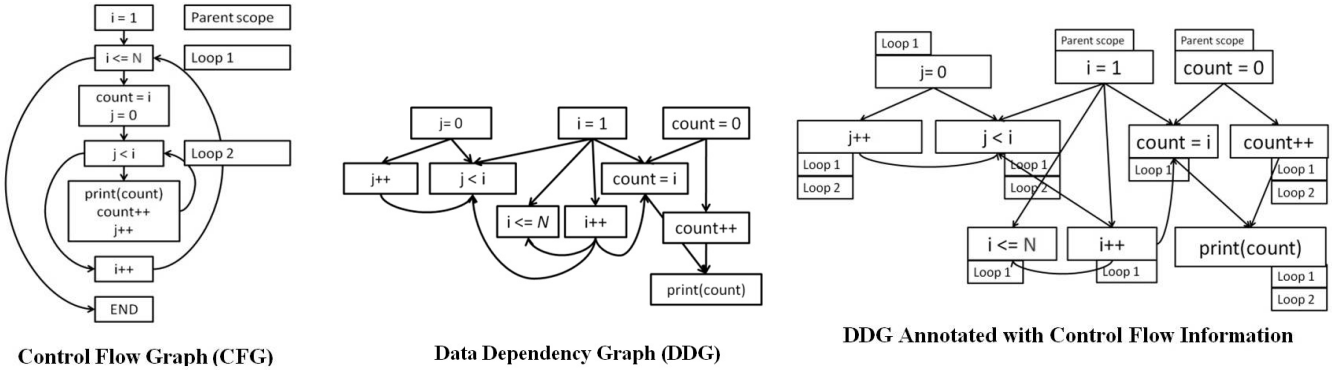

Fig. 1: Control & Data Information of Sample Program

nested-loop and one occurrence (in i++) within a single-loop. This class of features would correlate to Score 3 in the rubric (Table II)

d. Data-Dependency Features: This class includes features obtained by counting the occurrence of particular kinds of expressions which are dependent on other particular kinds of expression. Here, an expression refers to the features mentioned in (*b*). A data dependency is defined as any hierarchical ordering observed in the Data Dependency Graph (DDG) of the program. Such an ordering between two expressions generally signifies that the value of a variable(s) in one expression influences the evaluation of the other expression.

As an example illustrating this feature in the sample program, the occurrence of an edge from the node i++ to $count = i$ in the DDG of the sample program (see Fig. 1) suggests that the increment on i would affect the value of the variable $count$ in the expression $count = i$. Similarly, the edge from i++ to $j < i$ suggests that the evaluation of the expression $j < i$ is dependent on the increment on j happening in the expression j++. It is to be noted that these counts are generic and are not tagged to any particular variable in the program.

As in the previous classes of features, this class of features too can be characterized by varying levels of details - the dependencies could be listed out for an expression as a whole or could be specifically listed out for each variable appearing in an expression. The depth of the dependency captured could also vary - the dependencies could be restrained to the most recent expression which affects the current expression or could be traced back to n dependent expressions. Additionally, the various details of what constitutes an expression, as discussed in (*b*), can be incorporated here. In our work, we count the most recent dependency between each variable appearing in an expression separately.

e. Data-Dependency in Control Context: This class includes features obtained by counting the data-dependency features mentioned in d. in the context of the control structures they appear in. These features capture the specific functionality of a given program and are derived through a control-context tagged data-dependency graph of the program (see Fig. 1).
For e.g. in the illustration that has been provided in d., the expression pertaining to edge i++ to $count = i$ would additionally be annotated with a $loop$ as it appears within a for loop and the expression feature pertaining to i++ would be annotated with a $loop\ in\ a\ loop$ as it appears within a for loop-in-a-for loop.

It must be noted that the features described in (*d*) and (*e*) are of critical importance as they help identify a program belonging to a crucial level of the rubric (Score 4 and Score 5, Table II). They identify whether a candidate has understood an algorithm and has made use of key data and control dependencies in implementing his/her understanding.

f. Other Features: In addition to the features extracted from the control and data information of a program, other metrics may be used to predict how good a given program is. Such metrics can include - the number of lines in the source code, metrics in graph theory pertaining to the generated Abstract Syntax Trees (AST), Control Flow Graphs (CFG) and/or Data Dependency Graphs (DDG) such as the number of vertices, the number of edges, the height of the tree, the number of in-edges and out-edges etc. Additionally, the number of test cases a source code passes could also be used as a feature.

All the features mentioned above can be extracted from a combination of either the AST of a program, its CFG, its DDG and its Program Dependence Graph. An implicit advantage which these structures provide is the independence of the feature extraction process with the compilability of the code. As long as the source code follows the grammar of the program strictly, these features are easily extractable. This generally is of great utility when a candidate is unable to complete his/her program in the specified time and submits a partial solution to the problem.

III. MACHINE LEARNING APPROACH

We cast the problem of automatic grading in the standard machine learning framework - where programs attempted for a given problem are rated by experts, following which a regression model is developed based on the proposed features.

Given that a large number of features are generated, which include sparse features, a feature selection (or feature clustering) step may provide better modeling generalization. Clustering features in the given problem domain is intuitive, since more than one feature or feature-combinations may represent the same functionality. This can be done using techniques such as Principle Component Analysis, Factor Analysis, k-Means or by Latent Semantic Analysis (LSA)- a popular technique in the essay-grading literature.[3]

One advantage of using unsupervised methods is being able to use and extract information from ungraded samples as well. On the other hand, techniques such as forward selection, best subset selection, ridge regression, ensemble modeling can be used for supervised feature selection. In the experiments which follow, we use some simple feature selection techniques followed by the regression techniques of linear ridge regression and kernel-based SVMs. Ridge regression shrinks the feature weights leading to implicit feature selection.

One-Class Modeling: The approach of using regression to build a model entails the requirement of a domain expert to be involved to rate the programs for each problem manually. In an attempt to do away or reduce this requirement, we also explore a novel way of posing this as a problem in one-class modeling. We do so by identifying high quality codes amongst the candidate submissions using an automatic technique which looks at the number of test cases passed, programming practices used and the complexity of the code (discussed in Section IV). If the feature set indeed captures the functionality of the code and mimics the rubric, then a simple distance from these identified high-quality solutions in the feature space could provide the right grade (after some scaling). Whereas this distance approach mimics the neighborhood approaches prevalent in one-class classification (even though it is not classification), other approaches such as one-class SVMs, density estimation methods, etc. may also be used [14], [15]. Moreover, the unsupervised clustering approaches mentioned previously may also be used to cluster these programs. In this work, we show a preliminary demonstration of using a simple one-sided, absolute distance measure to predict grades and judge the efficacy of using one-class modeling for the problem at hand.

IV. EXPERIMENTS

We wanted to investigate the following questions with our experiments:

- How accurately can a machine learning approach based on our novel feature set predict grades as compared to grades given by human assessors?
- Do features derived from keywords, control-structures and data-dependencies (see Section II) add additional value in grade prediction over and above test-case based prediction?
- Do data-dependency and control-flow features add value over basic counts in the prediction and if so, by how much?
- What is the potential of one-class classification techniques in predicting accurate grades for programs, if only a set of high quality programs are available?

To answer these questions, we conducted experiments on two programming problems which were graded by domain experts. The problems were chosen such that the algorithms to solve them required different control-flow structures and data-dependencies to exist in implementations. We experimented with two machine learning techniques- Ridge Regression and SVMs, combined with different feature selection techniques. To test the efficacy of different feature sets, we built models by various combination of features. We also did preliminary investigations with one-class modeling techniques. We now discuss the details of the data sets used in the experiments.

A. Data Sets

We considered two programming problems for our experiments: the *Encrypt* problem and the *Alt-Sort* problem. Broadly, in the *Encrypt* problem, one has to convert a string into a new one by adding numbers to each character based on its position in the string. For the *Alt-Sort* problem, one has to sort a given list of numbers and return the alternating elements of the sorted list. Both the problems are above the beginners' level in programming and requires a fair amout of competence in algorithms.[4]

These problems, posed in the C programming language, were administered to over 550 senior-year undergraduate engineering students majoring in Computer Science or IT in India. They were delivered through *Automata*, an assessment product owned by Aspiring Minds[5], which provides an online compiler-based simulated environment. Each problem had a suite of test cases which checked basic and advanced conditions of the logic of the problems. The candidates had the option to edit, compile and check the correctness of their code on these suites as many times as they chose to before submitting final solutions.

Each program was rated on a scale of 1-5 following the rubric defined in Section II. Any program with no code or less than 5-6 lines of uncorrelated code was removed from the data set. After removing these, the data-set sizes for the *Encrypt* and *Alt-Sort* problems were 90 and 85 respectively. The data-set greatly reduced from what had been collected because a large number of candidates did not attempt the questions owing to poor programming ability. The ratings were done by two experts who had more than three years of experience in professional software development and were active participants in algorithmic programming contests. A consensus of the ratings of the two experts was taken. For programs where their grades did not match, they discussed the codes and reached a consensus.

The data-sets for each problem was split into two sets - training and validation. The train-set had 66% of the sample points whereas the validation-set had 33%. The split was done randomly taking care that the grade distribution in both the sets remained proportional. There were a total of 2045 features

[3]Such clustering may also be done using expert-knowledge as done through Wordnets etc. in Natural Language Processing [13].

[4]Given to a sample of computer science engineering students in India in their senior year, only 6.6% could solve them completely right.

[5]www.aspiringminds.in

TABLE IV: Regression Results - *Encrypt* Problem

Technique	Feature Type	Model Code	# Features	Cross-Val Error	Train r	Validation r
Ridge Regression	Test Cases	Enc-RR-TC	3	1.21	0.46	0.64
	Basic	Enc-RR-B	51	1.08	0.82	0.52
	All, w/o Test Cases	Enc-RR-AwTC	33	1.13	0.68	0.60
	All	Enc-RR-A	40	1.06	0.76	0.80
SVM (Linear Kernel)	Test Cases	Enc-SVM-TC	3	1.32	0.55	0.64
	Basic	Enc-SVM-B	20	1.10	0.52	0.53
	All, w/o Test Cases	Enc-SVM-AwTC	50	1.25	0.58	0.49
	All	Enc-SVM-A	47	1.25	0.63	0.49

TABLE V: Regression Results - *Alternate Sort* Problem

Technique	Feature Type	Model Code	# Features	Cross-Val Error	Train r	Validation r
Ridge Regression	Test Cases	Sort-RR-TC	3	0.84	0.79	0.76
	Basic	Sort-RR-B	24	0.85	0.84	0.61
	All, w/o Test Cases	Sort-RR-AwTC	47	0.80	0.89	0.78
	All	Sort-RR-A	37	0.67	0.95	0.90
SVM (Linear Kernel)	Test Cases	Sort-SVM-TC	3	0.89	0.76	0.77
	Basic	Sort-SVM-B	24	0.96	0.77	0.60
	All, w/o Test Cases	Sort-SVM-AwTC	50	0.85	0.81	0.81
	All	Sort-SVM-A	52	0.85	0.81	0.81

generated for the *Encrypt* problem and 594 features generated for the *Alternate Sort* problem.

B. Regression Modeling

We used feature selection followed by a regression with 3-fold cross-validation to model the grades. We used linear ridge-regression and SVM-regression with different kernels to build the models. The least cross-validation error was used for selecting the model. We used some simple techniques for feature selection, which included choosing the most correlating features, features which were most represented in the sample programs and the most correlating/represented features in each class of features.[6] The method of selecting the most represented features gave best results (cross-validation error). We swept over a threshold to vary the number of features used in building a model.

Experiment parameters: For linear regression with ridge regularization, we selected the optimal ridge coefficient λ by varying it between 1 to 1000 and selecting the parameter which gave us the least RMS error in cross-validation. For Support Vector Machines, we tested three of the primary kernels: linear, polynomial (3rd degree) and radial basis function. In order to select the optimal SVM, we varied the penalty factor C, parameters γ and ε, the SVM kernel and selected the set of values that gave us the lowest RMS error in cross-validation. We used an implementation of LIBSVM.[12] In the feature selection step, the number of features selected was varied from selecting those which appeared at least in 5% of the data-set to at least 50% of the data-set.

[6]We tried PCA to cluster features which did not yield good results. In this work, we did not attempt other sophisticated clustering of features due to very small number of samples as compared to the feature set size. This is discussed in Section V.

The experiments were done on four sets of features: first, a set of three test case features which captured the percent of basic, advanced[7] and edge cases the program passed; second, features pertaining to counts of keywords, tokens, expressions (without control context) and other metric-based features (see Section II); third, all features introduced in Section II and fourth, all features together with the test case features.

In the following subsections, the features pertaining to keyword and expression counts without control context are referred to as basic features while those pertaining to data-dependencies (including control context) are referred to as *advanced* features. These features collectively are referred to as *semantic* features.

C. Observations

The results of the experiment for both linear regression with regularization and SVM are tabulated in Table IV and Table V. These are results for the models selected according to least cross-validation error. We report the Pearson Correlation Coefficient (r) for the different models. The number of features used in these models is mentioned in Column 4. The best cross-validation error in case of SVM was produced by the linear kernel.

The observations are as follows: Ridge regression largely shows better cross-validation error and validation results than SVM regression. Thus, for further analysis, we consider the results of ridge regression. For both the problems, we see that the set of semantic features (models *Enc-RR-A* and *Sort-RR-A*) does better than the set of basic features (models *Enc-RR-B* and *Sort-RR-B*). The difference in performance is large, i.e.

[7]Advanced Cases contain pathological input conditions which would attempt to break codes which have incorrect / semi-correct implementations of the correct logic or incorrect / semi-correct formulation of the logic.

0.28 correlation points for *Encrypt* problem and 0.29 for the *Alt-Sort* problem. To study this further, we analyze the type of features selected in the models *Enc-RR-AwTC* and *Sort-RR-AwTC* (all features but test-cases) (see Table VI). While 35% and 60% belong to *basic* features, we see a good number of *advanced* features used too. The significant improvement in correlation value and use of other features in the model show the utility of the innovative features developed.

TABLE VI: Class-wise selected features

Model Code	# Test-Case	# Basic	# Adv	Total
Enc-RR-A	3	20	17	40
Enc-RR-AwTC	0	20	13	33
Sort-RR-A	1	12	24	37
Sort-RR-AwTC	0	16	31	47
Total Features				
Encrypt	3	116	1926	2045
Alt. Sort	3	57	534	594

Additionally, in both the problems, we see comparable performances between test-case models (models *Enc-RR-TC* and *Sort-RR-TC*) and models using semantic features (models *Enc-RR-AwTC* and *Sort-RR-AwTC*). This shows that the proposed features by themselves are able to extract information comparable in magnitude to that by test-case features, though they need not necessarily pertain to the same information. This is illustrated in models built using both, semantic features and test-cases (models *Enc-RR-A* and *Sort-RR-A*), which show better r than test-cases-only and semantic-features-only models - there is an improvement of 0.16 correlation points for the *Encrypt* problem and 0.13 for the *Sort* problem. It may thus be concluded that the use of our new feature set considerably improves the accuracy for the problem of grading computer programs.

However, there appears a difference in the validation results between the two problems. This could be attributed to the fact that sorting a list is a widely known algorithm and considerable variations in the implementation of the *Encrypt* problem can be expected, thereby allowing for varying results.

TABLE VII: Deviation of Predicted Score : Regression

Problem	% Shifts vs. # of Codes				
	#Val	$0-10\%$	$10-20\%$	$20-25\%$	$>25\%$
Encrypt	29	13	10	4	2
Alternate Sort	28	18	8	1	1

We now analyze how well our best models (models *Enc-RR-A* and *Sort-RR-A*) do in case they were used in a real-world scenario. In the validation set, we find the deviation of predicted scores from their corresponding expert-assigned scores. As seen in Table VII, we find that only 2 (6.9%) codes have more than a 20% deviation (which corresponds to a shift of one score in the range 1-5 of the rubric) for the *Alt-Sort* problem and 6 (21.42%) for the *Encrypt* problem. However, 4 out of these 6 codes for the *Encrypt* problem are within 25% deviation. We also discretized the scores to map them to a level of the rubric. The discretization was done by learning thresholds on the training-set for each score level such that the distribution of the predicted-scores vs. actual scores was the same. For the programs in the validation-set, the confusion matrix of the expert-assigned scores versus the discretized predicted scores are provided in Table VIII.

TABLE VIII: True vs. Discretized Predicted Scores

	Encrypt						***Alternate Sort***					
	Predicted Scores						**Predicted Scores**					
	1	**2**	**3**	**4**	**5**		**1**	**2**	**3**	**4**	**5**	
1	3	0	0	0	0	1	4	2	0	0	0	
2	0	4	1	1	0	2	0	5	4	0	0	
3	1	0	3	2	0	3	0	0	4	2	0	
4	0	0	2	10	0	4	0	0	1	3	0	
5	0	0	0	2	0	5	0	0	0	3	0	

We find that only one code has more than 1 level shift for the *Encrypt* problem, whereas there are none for the *Alt-Sort* problem. On the other hand, there is more confusion in the neighboring levels for the *Alt-Sort* problem as compared to the *Encrypt* problem. For both problems, we observe some range restriction towards the ends of the score-range, with no codes getting a score of 5. Over-estimation in the lower range and under-estimation in the higher range seems to be an artifact of linear regression.

These results indicate that our approach is robust enough to be used in applications where a one-level shift is acceptable. One finds similar correlation values and confusion matrices in automatic scoring of essays wherein such methods have found their way into high-stake assessments [16]. Likewise, such confusion between adjacent levels may also be observed between two expert raters. To study this further, we would have to find correlation and confusion between multiple expert raters and also see if the automatic score has better correlation with their consensus. On the other hand, more sophisticated machine learning techniques need to be further experimented.

D. One-Class Modeling

We do a very preliminary investigation of using only a set of high-quality codes for the purpose of prediction. The idea is to automatically find high-scoring codes to build a predictive model thereby reducing the effort invested in hand-grading programs. We identified a set of high scoring codes for both the problems in the following way- we took a subset of codes which passed more than 80% of the test-cases, solved the problem in the right time complexity (which was automatically determined by using an internally developed tool[8]) and followed programming best practices (such as lack of dead-code, etc.). This conservative filtering yielded a set of 27 codes for the *Encrypt* problem and 12 codes for the *Alt-Sort* problem. Their overlap with their expert-assigned scores is provided in Table IX.

Given such a small set of programs, the use of machine learning techniques was impeded. Thus, to predict scores, we used a simple one-sided distance metric, in which having less of a feature was penalized whereas having more was not. This was done with the intuition that having more of a particular feature was generally not indicative of an incorrect

[8]Broadly, the tool measures the run-time of the code for various input sizes and empirically derives the complexity.

TABLE IX: True Scores of Filtered Programs : One-Class Approach

Problem	True Scores					
	1	2	3	4	5	Total
Encrypt	0	0	0	22	5	27
Alternate Sort	0	0	0	2	10	12

code. We kept the same weight for all features, which could have potentially been learnt had there been enough data. This method is akin to neighbourhood approaches in one class classification. We provided scores to the programs based on the distances calculated from just the *basic* features and from all the semantic features (including the test-case features). The correlations between the calculated distance and the expert-assigned scores are tabulated in Table X. We provide correlations with and without including the train-set from which the distance is measured. Using the train-set here is not entirely incorrect in this context since it is automatically determined (as opposed to usual machine learning applications) and has been graded automatically, mimicking a real-world scenario which the system shall encounter. We find that without including the train-set, the correlation rivals test-case based models learnt in a supervised environment and betters it when including the train-set. Secondly, the advanced features add non-trivial value as compared to just using basic-features. This not only shows implicit correlation of our features to the rubric levels, but also shows great promise for one-class classification using the proposed features. More sophisticated techniques can be used which consider outlier data and unlabeled data to make better predictions.

TABLE X: Correlations : One-Class Approach

Problem	Without Training		With Training	
	Basic	All	Basic	All
Encrypt	0.60	0.65	0.68	0.75
Alterante Sort	0.56	0.69	0.64	0.80

V. CONCLUSION

The present work proposes the first machine learning based approach for automatic evaluation of computer programs. To facilitate the same, we design an objective rubric and a novel set of features that capture the program's functionality. These features are mined from abstract representations of the program such as the AST, CFG, DDG and the newly proposed control-context annotated DDG. Based on the rubric and hand-grades, we learn a prediction model using both regression and one-class modeling. We show that regression can provide much better grading than the ubiquitous test-case-pass based grading and rivals the grading accuracy of other open-response problems such as essay grading. We also show that our novel features add significant value over basic keyword/expression counts. Our preliminary investigations in one-class modeling show great promise and also indicate implicit correlation between the proposed novel features and the rubric.

This is the first step towards using machine learning for automatic grading of computer programs. We wish to extend this work with larger data sets and diverse problems, which could be facilitated by the growing interest in MOOCs and computer-based assessment systems. Not only can MOOCs facilitate scaling of such data-based evaluation methods, but the technique can help MOOCs on programming and algorithms truly scale by providing real-time feedback to candidates. A larger set of problems will help find out the generalizability of our technique. We also plan to use a more sophisticated mix of unsupervised and supervised learning techniques to develop better models for grading. The holy grail shall remain to uncover efficient one-class modeling techniques with a selection of implicitly correlating features to significantly reduce the effort of manual grading of programs.

ACKNOWLEDGMENT

The authors would like to thank Vinay Shashidhar for his help in certain aspects of this work and his invaluable suggestions.

REFERENCES

[1] C. Douce, D. Livingstone, and J. Orwell. *Automatic test-based assessment of programming: A review.* J. Educ. Resour. Comput., 5(3), Sept. 2005.

[2] M. Wick, D. Stevenson, P. Wagner, *Using testing and JUnit across the curriculum*, ACM SIDCSE Bulletin 37(1) (2005) 236-240

[3] Urs. Von Matt, *Kassandra: the automatic grading system*, SIGGUE 22(1994) 26-40

[4] M. Luck, M.Joy, *The BOSS system for On-line submission and assessment*, Computers and Education 25(3) (1995) 105-111

[5] G. Michaelson, *Automatic Analysis of functional program style*, Software Engineering Conference, Melbourne, Vic, IEEE(1996), 38-46

[6] K. Ala-Mukta, T. Uimonen, H.M. Jarvinen, *Supporting students in C++ programming courses with automatic program style assessment*, Journal of Information Technology Education 3 (2004) 245-262

[7] N. Truong, P. Roe, P. Bancroft, *ELP - a web environment for learning to program*, The 19th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education, Auckland, New Zealand, 2002, pp. 661670

[8] Milena Vujosevic-Janicic, Mladen Nikolic, Dusan Tosic, and Viktor Kuncak, *Software verification and graph similarity for automated evaluation of students assignments*, Information and Software Technology, 2012

[9] Wang, T., Su, X., Wang, Y. et al., *Semantic similarity-based grading of student programs*, Information and Software Technology, 49(2), 99-107

[10] R. Singh, S. Gulwani, and A. Solar-Lezama, *Automated feedback generation for introductory programming assignments*, In PLDI:Programming Language Design and Implementation (to appear) 2013.

[11] Sitthiworachart, J., and Joy, M., *Web-based Peer Assessment in Learning Computer Programming*, The 3rd IEEE International Conference on Advanced Learning Technologies: ICALT03, Athens, Greece, 9-11 July 2003

[12] C.C. Chang and C.J. Lin, *LIBSVM: A Library for Support Vector Machines*, 2001, Available: http://www.csie.ntu.edu.tw/~cjlin/libsvm

[13] A. Hotho, S. Staab and G. Stumme, *Wordnet improves text document clustering*, In Proceedings of the Semantic Web Workshop at SIGIR'03

[14] M. Breunig, H. Kriegel, R. Ng, and J. Sander, *LOF: Identifying density-based local outliers*, ACM SIGMOD Record, vol. 29, no. 2, pp. 93-104, 2000

[15] S. Papadimitriou, H. Kitagawa, P. Gibbons, and C. Faloutsos, *LOCI: Fast outlier detection using the local correlation integral*, in Proceedings of the 19th International Conference on Data Engineering, Bangalore, India, March 2003, pp. 315-326

[16] C. Leacock and M. Chodorow, *C-rater: Automated Scoring of Short-Answer Questions*, Computers and the Humanities, 37(4):389-405, 2003.