A System to Grade Computer Programming Skills using Machine Learning

Shashank Srikant
Aspiring Minds
shashank.srikant@aspiringminds.in

Varun Aggarwal
Aspiring Minds
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 system to grade computer programs automatically. In addition to grading a program on its programming practices and complexity, the key kernel of the system is a machine-learning based algorithm which determines closeness of the logic of the given program to a correct program. This algorithm uses 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. We show that the regression models 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 and report encouraging preliminary results. We show the value of the system through a case study in a real-world industrial deployment. To the best of the authors' knowledge, this is the first time a system using machine learning has been developed and used for 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.

Categories and Subject Descriptors: H.4 [**Information Systems Applications**]: Miscellaneous; I.2.6 [**Learning**]: General; K.3.0 [**Computers and Education**]: General

General Terms: Machine Learning; Program Analysis; Auomated Assessment

Keywords: Recruitment; Automatic grading; MOOC; Feature engineering; Supervised learning; One-class learning

KDD'14, August 24–27, 2014, New York, NY, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2956-9/14/08 ...$15.00.
http://dx.doi.org/10.1145/2623330.2623377 .

1. INTRODUCTION

Automatic evaluation of computer programming skills is a topic of keen interest today. It adds immense value to the recruitment processes of software development companies and also to teaching programming in training institutes, universities and Massively Open Online Courses (MOOCs). In the recruitment of software engineers, companies generally ask an applicant to solve a couple of programming problems which are subsequently discussed in an interview. The interviewer ascertains the 'closeness' of the candidate's algorithm to the correct solution and considers parameters such as programming practices used (e.g., dead-code, unused variables, etc.) and the time/space efficiency of the code to make a decision on the programming ability and algorithm design capability of the applicant. This is a non-standardized, time-consuming process and the interviewers are high-cost resources. The automatic assessment of codes can not only reduce the interview load, but also provide a report to the interviewer to assist him/her during the interview in highlighting important aspects of the candidate's performance. This would not only save time and money, but would also potentially open job opportunities to a larger set of applicants, who are missed out currently due to a lack of automated means.

In an education scenario, the benefits of such an automatic system are aplenty - first, 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. Second, 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 programs in order 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. Third, 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 of reference.

A primary aspect of such an automated grading system is the ability to grade a computer program on the basis of a rubric.[1][2] Such a rubric typically maps a score (quantitative measure) to the ability of the programmer to solve a 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 corporations and academia in making a sound judgment of a candidate's ability and helping him/her to write better programs. For instance, an applicant at the highest level of the rubric may be hired directly for the job of a software engineer, whereas someone at an intermediate level can be hired as a trainee and be deployed on projects after a stipulated training period in or outside the corporation.[1]

The most widespread approach currently used for automatic assessment of programs is the evaluation of the number of test-cases they passed[3][4][5][6]. 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 making them eventually fail the suite of test-cases designed for the problem[2]. 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[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.[3] 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 have designed an automated system to grade computer programs and produce a detailed feedback report for the student/interviewer. Firstly, the system grades how close the program's logic is to the correct solution, based on a rubric, using a novel machine learning approach. This machine learning approach, based on highly-informative features derived from abstract representations of a given program, is the subject of the present paper. Secondly, the system provides a score on the programming practices used in the program based on a rule-based system (rules as in [12]). Thirdly, it automatically detects the complexity of the program empirically by running the program for inputs of different sizes, recording the time to run and fitting a model to it. Based on these measurements, a comprehensive report is generated for each submitted program[13].

Specifically, the paper makes the following contributions:

[1]In India, large IT services companies hire 5000 to 10,000 software trainees every year, who undergo a three month training before deployment.

[2]In our own work, we see 40-50% programs falling in this category. Refer to Section 5 for details.

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

- We demonstrate the first machine learning approach to automatically grade programs based on a rich set of features.

- 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 2 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 available an unbalanced or a small number of graded programs.

- We demonstrate, by running the algorithm on a sample of responses for programming problems and a case study, the practical usefulness of the proposed technique and program evaluation system.

The paper is organized as follows: Section 2 introduces a rubric to grade programs and proposes a grammar to generate informative features from a program. In Section 3, we discuss a machine learning framework to solve the problem. In Sections 4, 5, we present the experiment details, case study of a real-world implementation and results. Finally, Section 6 discusses the results and future work.

2. FEATURES FOR PROGRAM GRADING

2.1 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, say an interviewer, 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 1 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

```
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 1: Pattern-printing problem with a sample program

A human evaluator going through the sample program mentioned in Table 1 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 2: 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.

2.2 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 2. 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.

2.3 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 2) 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

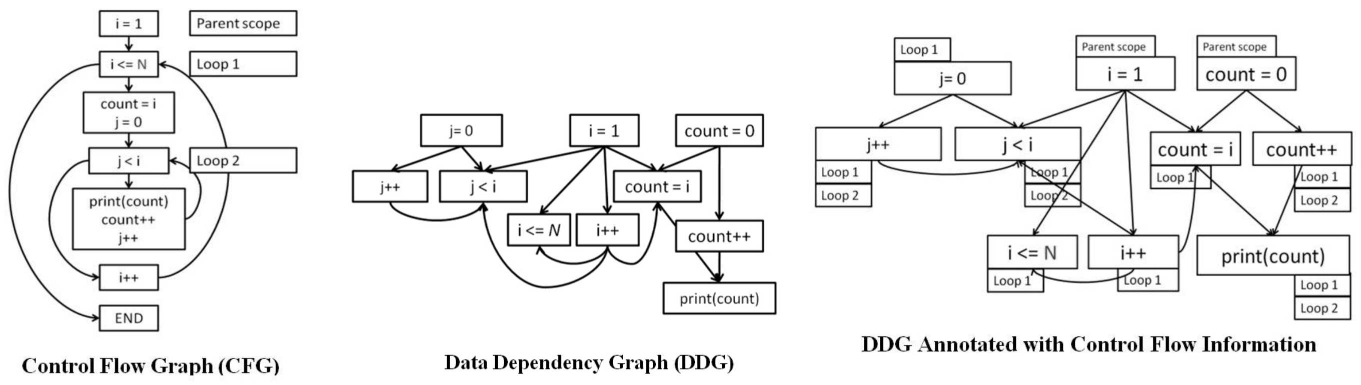

Figure 1: Control & Data Information of Sample Program

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 3 illustrates a few examples of expression-features and provides their significance.

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 nested-loop and one occurrence (in i++) within a single-loop. This class of features would correlate to Score 3 in the rubric (Table 2)

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 expressions pertaining to the edge i++ to *count* $= i$ would additionally be annotated with a *loop* for the *count* $= i$ expression as it appears within a for loop and the expression

Table 3: 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*:< $\uparrow$ *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* $\uparrow$ *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

feature pertaining to i++ too would be annotated with a *loop* as it appears within 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 2). 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.

3. 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.[4]

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, random forests 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 5). 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 [17]. 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.

4. SYSTEM DESIGN

We designed *Automata*, a delivery and evaluation system for testing computer programming skills. It provides an on-

[4]Such clustering may also be done using expert-knowledge as done through Wordnets etc. in Natural Language Processing [16].

line compiler-based simulated environment for applicants to code. Generally, a test contains two programming problems. Each problem has a suite of test cases which checks basic and advanced conditions of the logic of the problems. The candidates have 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. Based on the programs of the candidate, a detailed report is generated[13]. The report provides a total score and one score each on *Programming Ability* and *Programming Practices*. The first score is based on the grade of the program on the rubric (based on machine learning) and also on its empirically detected time complexity. The second score is based on a rule-based system (rules similar to [12]) that identifies bad programming practices. A final total score is provided, which is an average of these scores. *Automata* has been taken by more than 200,000 students and currently used by 20+ companies for recruitment purposes.[5]

5. 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 2) 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?
- What is the practical advantage of such a system in a real-world recruitment scenario?

To answer these questions, we conducted experiments on five 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 three machine learning techniques- Ridge Regression, SVMs and Random Forest, combined with different feature selection techniques. To test the efficacy of different feature sets, we built models by various combinations 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.

5.1 Data Sets

We considered five programming problems, named in the first column of Table 4. Broadly, in *Encrypt*, one has to add numbers to each character based on its position in a string; in *Alt Sort*, one has to sort a given list of numbers and return the alternating elements, in *Find Digit*, one is given a multi-digit number and a digit and s/he has to find the number of times the digit appears in the number; in *List Primes*, one has to list out all the prime numbers less than a given number and in *Print Spiral*, one has to print N lines of a spiralling pattern of digits. The problems are above the beginners' level in programming and requires a fair amount of competence in algorithms.

We used a sample set of programs written in *C* solved by senior-year undergraduate engineering students majoring in Computer Science in India. Each program was rated on a scale of 1-5 following the rubric defined in Section 2. Any program with no code or less than 5-6 lines of uncorrelated code was removed from the data set. 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 number of graded samples for different problems varied from 84 to 294 (mentioned in first column of Table 4)[6].

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. The total number of features generated for each problem are provided in Table 6.

Table 4: Performance across models - All features

Problem	Model	# feat	CV Correl	Val r
Encrypt	Ridge	80	0.85	0.79
(N = 106)	SVM - RBF	440	0.70	0.10
	Random Forest	87	0.96	0.29
Alt Sort	Ridge	68	0.93	0.91
(N = 84)	SVM - Linear	138	0.94	0.45
	Random Forest	56	0.98	0.68
Find Digit	Ridge	193	0.91	0.90
(N = 235)	SVM - Linear	164	0.96	0.76
	Random Forest	56	0.99	0.86
List Primes	Ridge	66	0.90	0.90
(N = 280)	SVM - RBF	85	0.90	0.60
	Random Forest	172	0.99	0.82
Print Spiral	Ridge	87	0.81	0.82
(N = 294)	SVM - RBF	123	0.90	0.60
	Random Forest	129	0.98	0.81

*N : Sample size for the problem

5.2 Regression Modeling

We used feature selection followed by a regression with 3-fold cross-validation to model the grades. We used linear ridge-regression, SVM-regression (with different kernels) and Random Forests to build the models. The best cross-validation correlation was used for selecting the model. We used some simple techniques for feature selection, which

[5]While all these candidates were tested on our delivery platform, they weren't necessarily graded by our machine learning approach. The deployment of machine learning was recent, happening at the time of writing this paper.

[6]Sample sizes are lower for the first two problems owing to their greater difficulty and removal of short/unrelated code samples

Table 5: Ridge Regression Results

Problem	Feature Type	Model Code	# Features	Cross-Val Correl	Train r	Validation r
Encrypt	Test Cases	Enc-RR-TC	3	0.39	0.39	0.54
	Basic	Enc-RR-B	60	0.62	0.87	0.41
	All, w/o Test Cases	Enc-RR-AwTC	35	0.57	0.72	0.56
	All	Enc-RR-A	80	0.61	0.85	0.79
Alt Sort	Test Cases	Sort-RR-TC	3	0.76	0.77	0.80
	Basic	Sort-RR-B	26	0.59	0.72	0.67
	All, w/o Test Cases	Sort-RR-AwTC	80	0.81	0.99	0.80
	All	Sort-RR-A	68	0.77	0.93	0.91
Find Digit	Test Cases	Digi-RR-TC	3	0.66	0.70	0.64
	Basic	Digi-RR-B	26	0.74	0.89	0.74
	All, w/o Test Cases	Digi-RR-AwTC	190	0.87	0.97	0.90
	All	Digi-RR-A	193	0.91	0.98	0.90
List Primes	Test Cases	Prime-RR-TC	3	0.74	0.75	0.80
	Basic	Prime-RR-B	35	0.83	0.88	0.69
	All, w/o Test Cases	Prime-RR-AwTC	134	0.85	0.91	0.82
	All	Prime-RR-A	66	0.90	0.94	0.90
Print Spiral	Test Cases	Spiral-RR-TC	3	0.83	0.83	0.84
	Basic	Spiral-RR-B	40	0.61	0.78	0.61
	All, w/o Test Cases	Spiral-RR-AwTC	166	0.66	0.81	0.64
	All	Spiral-RR-A	87	0.81	0.92	0.84

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. We swept over a threshold to vary the number of features used in building a model. We also experimented with randomized sparse models (using LASSO)[18] which is supposed to work well in cases with large number of highly correlated features. Its results consistently underperformed those of our simple techniques and are hence not reported for paucity of space. Our preliminary hypothesis for its underperformance is the high sparsity of our features.[7] The method of selecting the most represented features gave best results (cross-validation correlation of final 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 best cross-validation correlation. For Support Vector Machines [14][15], we tested three 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 best correlation in cross-validation. For Random Forests [14], we varied the number of estimators from 20 to 100 and allowed for bootstrapping. MSE was used a criterion to select between models. 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.

The experiments were done on four sets of features: first, a set of three test case features which captured the percent of basic, advanced[8] 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 2); third, all features introduced in Section 2 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.

5.3 Observations

The results of the experiment with different machine learning techniques are tabulated in Table 5. These are results for the models selected according to best cross-validation correlations. We report the Pearson Correlation Coefficient (r) for the different models. The number of features used in these models is mentioned in Column 3. The best cross-validation correlation in case of SVM was produced by the linear kernel for problems 2 and 3 and RBF kernels for problems 1,2 and 5.

[7]We tried PCA to cluster features which did not yield good results. We did not attempt other sophisticated clustering of features due to very small number of samples as compared to the feature set size, which we plan to do in future.

[8]Advanced Cases contain pathological input conditions which attempt to break codes implementing partially correct formulations of an algorithm.

Ridge regression consistently shows the best validation results and we consider its results (tabulated in detail in Table 5) for further analysis. We hypothesize that ridge regression's inductive bias rightly captures the problem space. For all the problems, we see that the set of semantic features (models with postfix -*AwTC*, Table 5, Column 3) does better than the set of basic features (models with postfix -*B*). The difference in performance varies from 0.13-0.16 correlation points (validation) for the problems, whereas it is 0.03 in case of *Print Spiral*. This clearly shows the utility of the advanced features in explaining the variance in human graders. To study this further, we analyze the type of features selected in the models with postfix *AwTC* (all features, without test-cases) (see Table 6). In all cases but one, the number of *advanced* features selected are more than the number of *basic* features selected, indicating their importance. The significant improvement in correlation values and considerable use of advanced features in the models demonstrate the utility of the innovative features developed.

Table 6: Class-wise selected features

Model Code	# Test-Case	# Basic	# Adv	Total
Enc-RR-A	1	39	40	80
Enc-RR-AwTC	0	24	11	35
Sort-RR-A	2	26	51	79
Sort-RR-AwTC	0	28	52	80
Digi-RR-A	3	44	146	193
Digi-RR-AwTC	0	44	146	190
Prime-RR-A	1	30	35	66
Prime-RR-AwTC	0	40	94	134
Spiral-RR-A	1	31	55	87
Spiral-RR-AwTC	0	43	123	166
Total Features				
Encrypt	3	164	4017	4184
Alt Sort	3	82	1586	1671
Find Digit	3	161	1269	1433
List Primes	3	266	1818	2087
Print Spiral	3	198	2117	2318

Additionally, we see competitive performance between test-case models (models with postfix -*TC*) and models using semantic features (models with postfix -*AwTC*). The models built using both, semantic features and test-cases (models with postfix -*A*) show better or equivalent *r* than test-cases-only and semantic-features-only models. The validation correlation for the models with all features range from 0.79-0.9 across the five problems and from 0.54 to 0.84 for test-cases. It may thus be concluded that the use of our new feature set considerably improves the accuracy for the problem of grading computer programs.

We now analyze how well our best models (models with suffix -*A*) do in case they were used in a real-world scenario. We find the deviation of predicted scores from their corresponding expert-assigned scores, in the validation set. As seen in Table 7, we find that more than 80% of the predicted grades are within a 1-grade shift from it's corresponding expert-rated grades. Only a maximum of three codes in every problem are predicted grades which are greater than 2-grade shifts away. 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. As an illustration, the confusion matrix of the expert-assigned scores versus the discretized predicted scores are provided for two of the five problems in Table 8 for the programs in their validation-set

Table 7: Deviation of Predicted Score : Regression

Problem	**Score Shifts vs. # of Codes**					
	#Val	0-1	1-2	2-3	3-4	>4
Encrypt	34	28	5	1	0	0
Alt Sort	27	23	4	0	0	0
Find Digit	77	72	4	1	0	0
List Primes	92	78	11	2	1	0
Print Spiral	97	79	15	2	1	0

Table 8: True vs. Discretized Predicted Scores

Encrypt					
	Predicted Scores				
	1	**2**	**3**	**4**	**5**
1	3	1	0	0	0
2	1	3	1	1	0
3	0	2	3	2	0
4	0	1	4	9	1
5	0	0	0	0	1

List Primes					
	Predicted Scores				
	1	**2**	**3**	**4**	**5**
1	20	3	2	0	1
2	2	5	1	1	0
3	0	1	8	1	1
4	0	0	2	5	4
5	0	0	0	2	33

We find that only two codes (5.8%) have more than 1 level shift for the *Encrypt* problem, whereas there are 4 (4.3%) for the *List Primes* problem. In *Encrypt*, 55.5% codes have the correct grades (owing to its lower validation correlation), whereas *List Primes* has 77.2% correct grades.

These results indicate that our approach is robust enough to be used in applications where a one-level shift is acceptable for a limited number of cases. One finds similar correlation values and confusion matrices in automatic scoring of essays wherein such methods have found their way into high-stake assessments[19]. 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.

5.4 Feature Insight

We wished to see what insight could be derived by looking at the features picked up by machine learning for a problem. This could not only help automatically discover important logic elements needed in a correct solution of a problem, thereby enabling us to give feedback/hints to students on what constructs to use while solving a problem, but also help design better features. As a preliminary step towards this goal, we show the most contributing feature for *Find Digit* from the high performing ridge-regression model.

$$\begin{array}{c} \text{Dep@Var:1,Op:!=,Const:1\#input:m\_LOOPc} \\ \uparrow \\ \text{Var:1,Op:/,Const:1\#input:m\_LOOPb} \end{array}$$

The pseudo-code equivalent of the above described feature is shown in Table 10, left column. The algorithm to

Table 9: Correlation - Distance metric

Problem	***All Features***						***Only Basic***					
	With training			*Without training*			*With training*			*Without training*		
	Mean	***Min25***	*Min*	*Mean*	***Min25***	*Min*	*Mean*	***Min25***	*Min*	*Mean*	***Min25***	*Min*
Encrypt	0.57	**0.61**	0.55	0.65	**0.65**	0.51	0.52	**0.56**	0.57	0.59	**0.63**	0.54
Alt Sort	0.80	**0.83**	0.83	0.80	**0.83**	0.83	0.72	**0.75**	0.73	0.68	**0.69**	0.70
Find Digit	0.75	**0.81**	0.84	0.72	**0.80**	0.77	0.59	**0.73**	0.75	0.56	**0.67**	0.67
List Primes	0.81	**0.81**	0.66	0.76	**0.78**	0.54	0.75	**0.75**	0.63	0.65	**0.66**	0.48
Print Spiral	0.68	**0.69**	0.72	0.58	**0.58**	0.52	0.55	**0.61**	0.62	0.49	**0.51**	0.48

solve *Find Digit* (see Section 5.1 for problem details) requires each digit of the multi-digit number to be extracted and checked for equality against the input digit. The feature describes a dependency condition in control context: the input (N) is used in the condition of a loop with a ! = operator and a constant, which depends on an assignment made to it within the body of the same loop (dependence becomes clear if one unrolls the loop), where it is assigned an expression using a / operator and a constant. Clearly, this is a general logical structure which occurs in one of the basic implementations for the problem as illustrated in the right column of Table 10. The automatic discovery of this structure, which humans will look for in a correct implementation, illustrates the power of the technique and also a first step towards getting qualitative insight into problem solutions through machine learning.

```
...                         int find_digit(int N, int digit){
LOOP(N! = const ){              while(N != 0){
    ...                             d = N %10;
    N = N/const                     if(d == digit)
    ...                                 ...
 }                                  N = N/10;
...                             }
                            }
```

Table 10: Most contributing feature - Pseudo code

5.5 One-Class Modeling

We do a 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 the problems in the following way- we took a subset of codes (referred to as *good set*) which passed more than 80% of the test-cases, solved the problem in the right time complexity and followed programming best practices. The number of codes this conservative filtering yielded is recorded in Table 11. Their overlap with their expert-assigned scores is also shown.

Given a small number of programs in the *good set* for some problems, the use of machine learning techniques was impeded. Thus, to predict scores, we decided to define the distance of a given code from the *good set* in the feature space. 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 code. We kept the same weight for all features, which

Table 11: True Scores of Filtered Programs : One-Class Approach

Problem	**True Scores**					
	1	**2**	**3**	**4**	**5**	**Total**
Encrypt	0	0	0	22	5	27
Alt Sort	0	0	0	2	10	12
Find Digit	0	0	3	15	130	148
List Primes	3	0	0	9	93	102
Print Spiral	0	1	0	1	149	150

could have potentially been learnt had there been enough data. Our method is akin to neighbourhood approaches in one class classification.

We provided scores to the programs based on the distances calculated using just the *basic* features and all the semantic features (including the test-case features). In order to calculate a score for a given program, we first sum the distance of the program from the good set. This provides a distance of the given program from each of the programs in the *good set.*We then employ three methods - a. consider the mean of these distances b. consider the minimum of these distances and c. consider the mean of the least 25% of the distances. We hypothesized that the mean metric shall be noisy given that the *good set* would have codes implementing different algorithm strategies. On the other hand, the minimum of the distances would be noisy given the presence of outlier codes (columns 1,2 and 3 of Table 11). The mean of the least distances is a good trade-off. Interpreting it differently, it attempts to identify the cluster (algorithm strategy) the current program belongs to and finds the distance from it.[9]

The correlations between the calculated distance and the expert-assigned scores are tabulated in Table 9. We provide correlations with and without including the train-set from which the distance is measured. Using the train-set 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. Akin to our observations in the supervised learning setting, we find that scores from distances across all features outperform those only from *basic* features. Secondly, we find $Min25$ to provide the most consistently high performing results. This indicates a validation of our hypothesis regarding the existence of clusters and outliers codes in the *good set.*

[9]In future, for problems with larger sample sizes, we intend to cluster codes in the *good set* using unsupervised clustering algorithms and find distances from the clusters.

5.6 Case Study

We studied the deployment of *Automata* at a hiring event of one of Aspiring Minds' large IT-product customers. The event had a turn-up of 1050 applicants for a software engineering job. They were tested on two programming problems (*List Primes* and *Encrypt* from above set) in a 75 minutes test. The customer's default shortlisting criterion was to consider candidates who cleared at least 80% of the test-cases.

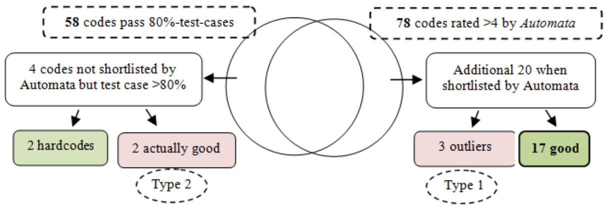
58 codes pass 80%-test-cases
78 codes rated >4 by Automata
4 codes not shortlisted by Automata but test case >80%
Additional 20 when shortlisted by Automata
2 hardcodes
2 actually good
3 outliers
17 good
Type 2
Type 1

Figure 2: Case study - Extant approach vs. *Automata*

Based on the test, candidates above a programming ability score of 78 were shortlisted, which maps to score 4 on the rubric. We studied how these results would have been affected had we used the test-case pass criteria (Fig. 2). For a test-case pass rate of 80%, 58 candidates would have been shortlisted. Among the 20 extra candidates, only 3 were outliers as inspected by the interviewers. This implies that the test-case criteria would have missed 22.6% ($\frac{17}{58+17}$) of the good candidates as compared to our approach.[10] On the other hand, there were 4 candidates who were accepted by test-case, but rejected by our scoring: two of these had hard-coded the logic in their implementations(thus rightly rejected), whereas, 2 of them were outliers. Of the final offers made, 45% more selections were made from the top 50% scoring students. This preliminary case-study demonstrates how our approach gives immediate benefit and provides guidance to interviewers by reducing type-2 errors.

6. CONCLUSION

The present work proposes a system to automatically grade computer programs using machine learning. To facilitate the same, we design an objective rubric and a novel set of features that capture the program's functionality. We show that regression against expert-grades 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. Finally, through a case study, we show the practical efficiency provided by a deployment of the system in the recruitment process of an IT-product company.

In future, as our data sets grow in size, we want to use more sophisticated unsupervised and supervised machine learning techniques. Given that supervised learning is hungry for graded data, we see programming classes in universities and MOOCs as an avenue to get teaching-assistants' grades effortlessly. Integration of this system into classrooms can facilitate a seamless continuous collection of data to build a powerful automated grading system. The holy grail shall remain to build problem independent grading techniques or those that reduce the need of graded data, which may be facilitated by efficient one-class modeling techniques. An interesting area to explore, preliminarily investigated in this paper, is the qualitative feedback from our techniques through feature analysis. Finally, we look forward to the large scale deployment of our system in the industry and academia to facilitate further learnings and improvements.

[10]In this case, we compare the performance of our results against the performance of test-cases and not the gold standard of expert-grades.

Acknowledgment

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

7. REFERENCES

[1] V. Aggarwal, S. Srikant, V. Shashidhar. Principles for using Machine Learning in the Assessment of Open Response Items : Programming Assessment as a Case Study. 2013, NIPS Workshop on Data Driven Education
[2] S. Srikant, V. Aggarwal. Automatic Grading of Computer Programs: A Machine Learning Approach. 12th International Conference on Machine Learning Applications (ICMLA), 2013.
[3] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. J. Educ. Resour. Comput., 5(3), Sept. 2005.
[4] M. Wick, D. Stevenson, P. Wagner. Using testing and JUnit across the curriculum. ACM SIDCSE Bulletin 37(1) (2005) 236-240
[5] Urs. Von Matt. Kassandra: the automatic grading system. SIGGUE 22(1994) 26-40
[6] M.Joy, N. Griffiths, R. Boyatt. The BOSS online submission and assessment system. Journal on Educational Resources in Computing 5(3) (2005)
[7] 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
[8] M. Vujosevic-Janicic, M. Nikolic, Dusan Tosic, and V. Kuncak. Software verification and graph similarity for automated evaluation of students assignments. Information and Software Technology, 2012
[9] T. Wang, X. Su, Y. Wang 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. Programming Language Design and Implementation (PLDI) 2013.
[11] J. Sitthiworachart, and M. Joy. 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] PC-Lint Software, Gimpel Software, www.gimpel.com
[13] *Automata* sample report, `http://www.aspiringminds.in/docs/sample_report.pdf`
[14] Pedregosa et al. Scikit-learn: Machine Learning in Python. JMLR '12, pp. 2825-2830, 2011.
[15] C.C. Chang and C.J. Lin. LIBSVM: A Library for Support Vector Machines. 2001, Available: `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
[16] A. Hotho, S. Staab and G. Stumme. Ontologies improve text document clustering. In Data Mining, 2003. Third IEEE International Conference on, ICDM 2003.
[17] 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
[18] N. Meinshausen, P. Buhlmann. Stability selection. Journal of the Royal Statistical Society, 72(2010)
[19] C. Leacock and M. Chodorow. C-rater: Automated Scoring of Short-Answer Questions. Computers and the Humanities, 37(4):389-405, 2003.