Grading uncompilable programs

Rohit Takhar and Varun Aggarwal
{rohit.takhar, varun}@aspiringminds.com
Aspiring Minds

Abstract

Evaluators wish to test candidates on their ability to propose the *correct* algorithmic approach to solve programming problems. Recently, several automated systems for grading programs have been proposed, but none of them address uncompilable codes. We present the first approach to grade uncompilable codes and provide semantic feedback on them using machine learning. We propose two methods that allow us to derive informative semantic features from programs. One of this approach makes the program compilable by correcting errors, while the other relaxes syntax/grammar rules to help parse uncompilable codes. We compare the relative efficacy of these approaches towards grading. We finally combine them to build an algorithm which rivals the accuracy of experts in grading programs. Additionally, we show that the models learned for compilable codes can be reused for uncompilable codes. We present case studies, where companies are able to hire more efficiently by deploying our technology.

1 Introduction

Evaluators often wish to test students and jobseekers on their ability to propose the *correct* logic, aka algorithmic approach, to solve a given programming problem. Evaluators care most about this skill rather than the ability to follow and reproduce the syntax of a language. Syntax can be learnt quickly for a new language. Thus, candidates are often asked to write the program in a language of their choice or even as a pseudo-code. Evaluators then examine the code and discuss to see if the stated algorithm (not the exact program) is correct and efficient.

Several automated systems for testing coding skills and providing feedback have been developed in the last decade. The traditional method for evaluation of the code uses test cases, which neither provides a good judgment to the evaluator nor proper actionable feedback to the candidate. Recently, several new methods of grading and feedback that use techniques such as machine learning and program analysis have been developed. In (Srikant and Aggarwal 2014), the authors present a system which provides a grade on the logical correctness of a program according to a rubric. It uses machine learning and control structure and data dependency features. In (Singh, Gulwani, and Solar-Lezama 2013; Head et al. 2017), the system automatically provides feedback on student programs by finding fixes. There are also tools (Le Goues et al. 2012; Long and Rinard 2016; Mechtaev, Yi, and Roychoudhury 2016) that fix logical bugs automatically in softwares.

None of these approaches handles codes that do not compile.[1] They require a compilable code to be able to grade them or provide feedback. Generally, codes that do not compile constitute 34% of the total submitted codes in MOOCs (Bhatia and Singh 2016) and 46% in applicant pool for coding jobs (our study). Candidates miss out on good grades or jobs, even when they write uncompilable codes with correct or near correct logic (see examples in Table 5). They are unable to get any feedback on their codes either. Evaluators miss on identifying or hiring potentially good recruits.

We present the first approach to grade uncompilable codes and provide *semantic* feedback on them. By semantic feedback, we imply feedback on the logic/algorithm of the code and not feedback on how to make the code compilable. Our feedback comprises whether the code has the algorithmic structures to solve a given programming problem, for instance, does it have the required control structures or data operations (refer to Table 4 for rubric definition). We intend to discover the *intent* of the coder — capture what she wished to imply had the code compiled. We do this by various ways — imposing less stringent grammar/rules than the compiler, skipping non-understandable parts and by fixing compilation errors. These methods allow us to derive rich semantic features from uncompilable codes. We then build machine learning (ML) based grading models using these features. Also, we investigate the possibility of reusing of available ML models developed using labelled compilable codes. We suggest methods to re-use these models with limited sample of labelled uncompilable codes.

Using our method, we find that 15.5% of uncompilable codes submitted by students seeking job with a China internet company had near correct logic, while 21.9% had the right control structures. In particular, the paper makes the following contributions:

- We present the first approach to grade and provide seman-

[1]There are approaches (Gupta et al. 2017; Bhatia and Singh 2016) to correct uncompilable codes to make them compilable, but they do not provide semantic feedback.

tic feedback on codes that do not compile. This is a first step to decouple the skill to propose the right algorithm for a programming problem vs. writing syntactically correct code.

- We present different novel methods to derive semantic features from codes that do not compile and compare their efficacy.
- We present methods to reuse machine learning based grading models built for compilable codes.

The paper is organized as follows — In Section 2, we discuss the machine learning approach to grade programs. In Section 3 and Section 4, we present the techniques to extract features from uncompilable programs and machine learning models. In Section 5, we discuss the details of our experiments and results. Section 6 discusses the deployment of our system in a high stakes assessment. Section 7 concludes the paper and discusses future work.

2 Grading programs using machine learning

In this section, we discuss the machine learning based approach proposed in (Srikant and Aggarwal 2014) to grade programs. It uses a set of features which are derived from abstract representations of a given program. These features capture the semantic relationships present in the program. The counts and relations of the following properties of a program are extracted — the keywords, the expressions, the control structures and the expression dependencies. The different features extracted from a program are as following:

- **Keywords:** Examples include counts of all keywords, tokens, operators etc. such as the number of times a '*' operator appears or loop and conditional tokens such as 'for' and 'if' appear.
- **Expressions:** Expressions such as $y = x\%2$ are abstracted to a notation such as $v : 2 :: op : \% :: c$:'2', which denotes an expression having two variables, one modulus operator and the constant '2'. The number of occurrences of such abstract expressions are counted.
- **Expression Dependency:** A data dependency is captured when the variable in one expression is used in another expression.[2] For example, the expression $x < y$, which contains a relational operator ($<$) and two variables, is dependent on the expression $y++$, which contains a post increment operator (++) and one variable. This is denoted using the notation $v : 1 :: op : ++ (\leftarrow) v : 2 :: op : relation$. The occurrences of each dependency matching such abstractions is counted. This is repeated for each unique pair of dependencies that appears in a response.
- **Control Context:** Separate counts are maintained for each of the three properties described above according to the control-context (loops and conditional statements) in which they appear. For example, an expression whose abstract notation matches $v : 2 :: op : \% :: c$:'2' is counted separately if it appears within an *if statement* as opposed to a loop like a *for* or a *while* as opposed to an *if statement* within a *for*.

[2]For a given variable, only expressions in its scope block are considered.

These features are transformed into a set of features that maintain their structural relation with the labels across programming tasks (Srikant and Aggarwal 2014). Using these transformed features a task-independent supervised model is learned across programming tasks. This model is used to predict grades for a new code. We require the Abstract Syntax Tree (AST) and the Symbol Table (ST) to generate these features. The code needs to compile for this. In the next section, we describe how we generate features for uncompilable code.

3 Extracting features from uncompilable codes

We wish to find the *intent* of the programmers even though they haven't followed the grammar/syntax rules of the programming language.[3] One may extrapolate, add or modify code, to make the program syntactically correct and anticipate the programmer's intention. Another option is *ignore* rules, that the code doesn't follow, interpret non-understandable snippets or *skip* them. We try both these methods.

Correcting the program: We use two methods to make the code compilable. The first is a rule based system, where common errors are fixed by say, inserting a semicolon, balancing parentheses, declaring undeclared variables, or adding a $return$ statement. We do multiple passes of these fixes, one at a time, until the code compiles.

In the second method, we predict the correct line at the position of compilation error based on previous keywords. We learn n-gram models on good codes for a particular program to make the prediction.[4] Generally, if the predicted line makes the program compilable, the statement is accepted. In our approach, we do not accept all such edits. We find the difference between the generated statement and the original one. We only use the generated statement for differences such as insertion of a data type, a variable name, etc. and not where new logical units, such as operators, expressions or control structures are introduced. We wish to grade the program only based on the logical units written by the programmer, and not introduce new ones.

We first run the rule-based system to fix the general errors. For codes that are still uncompilable, we run the n-gram based token prediction approach to remove complex errors. We run the updated programs again through the rule based system to correct any declaration/definition errors introduced by the n-gram approach.[5] We call this combined approach as the MC (Make Compilable) method. On average, it makes 48% codes in our Chinese data set compilable.

Relaxing the compilation rules: We require the AST and

[3]This is akin to the task of finding user intent in natural language understanding systems.

[4]We also tried an LSTM based RNN model, but it did not provide any better performance than n-gram on our data set. The approach is described in detail in (Bhatia and Singh 2016).

[5]This is due to insertion of variable/class name tokens which were not initially declared in the program.

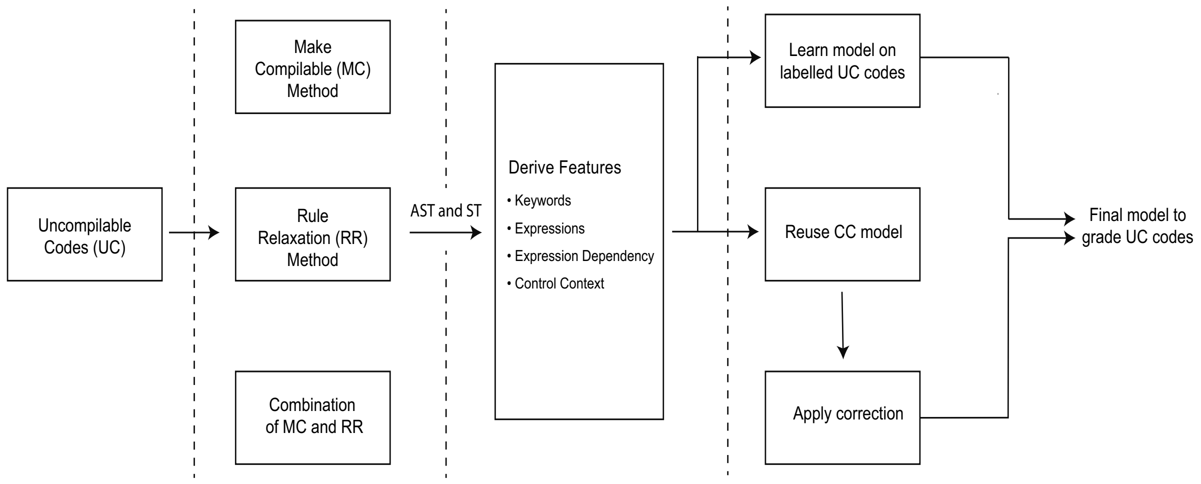

Figure 1: The flow to grade uncompilable programs with various possible alternatives at each step.

ST to extract our features. A typical compiler consists of several steps — tokenization, parsing based on a grammar, type checking, linking and optimization (Appel 1997). We loosen the rules in the various compilation steps to generate the AST and ST for an uncompilable program. We intend to:

(a) Capture the bulk of the correct statements

(b) Accept/reinterpret uncompilable statements by rule relaxation

(c) Skip those that are not parseable

The tokenization step doesn't generally lead to an error. Non-meaningful tokens get identified as potentially variable/function names, to be disambiguated later. For instance, an undeclared variable, or a variable with an operation not matching its data type, are accepted at this stage. A spelling error in a token, say $rturn$ in $rturnflag$ is wrongly identified as a class identifier, whose definition will be expected to be linked in later steps.

In the parsing step, errors happen when the code violates the grammar rules. We skip such statements. For instance, the statement, $int\,[]\;arr = new\,int[]$; gets skipped. We build an *Imperfect AST* in the parsing steps — it captures the good part of the code, while uncompiling pieces are skipped. On average very few tokens are skipped, around 6.2% for our sample.

The type checking/linking steps generate errors in case operations/declarations of variables do not match; for missing declarations/definitions of variables, functions and so on. We ignore wrong declarations and assume missing declarations. This constitutes our relaxation of the rules. In the previous examples — the operation is accepted even if the data type doesn't match, so is usage of undeclared variables and $rturn$ is considered as a class. We create an *Imperfect ST* for the program. We call this the RR (relax rules) method.

Each of these approaches have their relative advantages. The first is better at inferring the intent of the coder by correcting the non-understandable statements. However, it fails when it cannot correct — 52% times for our sample. The second approach skips or even misconstrues what it cannot understand. However, it parses 95% of the codes.

Table 1: Sample size for different programming tasks

Task Name	Sid	#UC	#CC
countCacheMiss	24	370	106
balancedParentheses	132	355	70
grayCheck	43	367	175
transposeMultMatrix	48	407	182
eliminateVowelString	62	392	182

4 Machine Learning Models

Figure 1 shows the final flow for our algorithm. We start with a set of labelled uncompilable (UC) codes. We generate features from these codes based on one of the following: MC method, RR method or a combination of both. The combination method is motivated and described in detail in Section 5. We then develop a machine learning model using these features.[6] In the first case, we perform supervised learning using the UC codes to predict the expert grades (labels). In the second, we reuse models already trained on labelled compilable codes (CC). We report in the next section, that the models (trained on CC) do not perform well in themselves. However, certain corrections to them make them as good as those trained on the labelled UC code sample.

We developed machine learning models based on the responses collected on the programming tasks listed in Table 1. We report results for *LASSO* ($\alpha = 1$) (Tibshirani 1994)

[6]The modelling techniques are described in detail in the next section.

Table 2: Accuracy of models built using the RR and MC approach to grade uncompilable codes. Metrics: r and MAE.

			RR (all)		MC		RR (MC set)	
Sid[1]	Sample Size	Sample Size (MC)	r	MAE	r	MAE	r	MAE
24	370	171	0.71	0.52	0.78	0.44	0.71	0.51
132	355	165	0.70	0.43	0.63	0.45	0.65	0.44
43	367	175	0.72	0.47	0.86	0.25	0.78	0.49
48	407	220	0.66	0.50	0.65	0.54	0.72	0.43
62	392	198	0.59	0.65	0.74	0.55	0.58	0.68
Mean			**0.68**	**0.52**	**0.73**	**0.45**	**0.69**	**0.51**
Median			**0.70**	**0.50**	**0.74**	**0.45**	**0.71**	**0.49**

[1] Some data from SIDs in bold was used for training purposes. In this table we only report results for test data on all SIDs.

we varied λ from 0 to 4.[7] The model which gave the best cross-validation (three-fold) correlation was selected.
We develop generalized task-independent models (detailed in (Singh, Srikant, and Aggarwal 2016)). Thus, we trained on data from three tasks (Sid 43, 48, 62), and tested on data from all five tasks. For the tasks used in training, we split the response set into a 70-30 train-test set. For the other two tasks, all responses were used in the test set. The performance of the model on the unseen problem set helped demonstrate how well the models generalized to programming tasks whose sample was not used in training.
We report the Pearson correlation coefficient (r) and MAE ($\sum \frac{|ypred-y|}{n}$) as evaluation metrics to judge the performance of our models.
Now, we describe our experiments, which compare the efficacy of various approaches based on choices made at different steps.

5 Experiments and Results

Our experiments were designed to address the following questions:

- Which among MC and RR approaches are more accurate in predicting the grade for uncompilable codes?
- How accurately can we grade uncompilable codes? How does it compare to the accuracy of expert grades and automatic grading of compilable codes?
- Can we re-use models trained for compilable codes for uncompilable codes?

We conducted our experiments on five programming tasks. The tasks were chosen such that the algorithms to solve these tasks had varying complexities. A subset of programs written by candidates for these tasks were graded by experts. The set included both compilable and uncompilable codes. We built separate ML models for the compilable and uncompilable codes to predict expert grades. For the uncompilable codes, we built models using each of the RR and MC approach and compared their accuracy. We further investigated if the models for CC could be reused for predicting grades for UC. We now discuss further details.

[7]We used linear regression, linear regression with L_1 regularization (LASSO), linear regression with L_2 regularization (Ridge regression), decision trees, random forests and SVMs. Similar to what was seen in (Singh, Srikant, and Aggarwal 2016), linear models worked the best among all these techniques, indicating linearity in the inherent structure of this problem space. We report results only for LASSO in this work which outperformed all other techniques.

Data

The experiments were run on a set of programming tasks hosted on *Automata*, our automated programming evaluation platform (Aspiring Minds 2012). Respondents, who were college seniors majoring in computer science, took a 90 minute assessment in a proctored environment wherein they attempted two programming tasks. For our experiments, we considered five tasks and programs written in Java (see Table 1). The topics covered by these programming tasks spanned iterative/ recursive algorithms, trees and graphs and other algorithms like the shortest job first, etc. We used on an average, 143 compilable and 378 uncompilable responses per task to build and test our models. In total, 2606 codes were used in our experiments.
Two professional software engineers with 4-7 years' experience each shared the task of grading the responses. The experts followed the rubric defined in to grade codes on a scale of 1-5 (described in (Srikant and Aggarwal 2014)). Before beginning the grading exercise, they underwent a one-week workshop wherein they learned how to interpret the rubric and participated in mock grading exercises. The experts were given special instructions to grade the codes only based on the intended logic of the coder and not to penalise based on the quantum or type of compilation errors. The correlation between the grades of the two experts was on an average 0.72 across the programming tasks in the data set.

Results

To answer the first question, we trained models on the uncompiling codes using each of the RR and MC approach. The MC approach couldn't correct all the codes. For an apple to apple comparison, we also present RR results for the

Table 3: Accuracy of models built using a sample of compilable and uncompilable codes to grade uncompilable codes. Metrics: r and MAE.

	CC Model		CC Model with dist. Fixed		CC Model w/o TC and dist. Fixed		Uncompilable	
Sid[1]	r	MAE	r	MAE	r	MAE	r	MAE
24	0.74	0.75	0.74	0.47	0.74	0.44	0.75	0.48
132	0.75	0.84	0.75	0.41	0.74	0.41	0.68	0.45
43	0.72	0.68	0.72	0.47	0.79	0.46	0.79	0.43
48	0.60	0.77	0.60	0.65	0.64	0.60	0.62	0.56
62	0.56	1	0.56	0.58	0.64	0.52	0.69	0.57
Mean	**0.67**	**0.81**	**0.67**	**0.52**	**0.71**	**0.49**	**0.71**	**0.50**
Median	**0.72**	**0.77**	**0.72**	**0.47**	**0.74**	**0.46**	**0.69**	**0.48**

[1] Some data from SIDs in bold was used for training purposes. In this table we only report results for test data on all SIDs.

subset of codes corrected by the MC approach. These results are provided in Table 2. MC is able to only grade 48% of codes, but performs better than RR on this set. The mean (median) r for MC is 0.73 (0.74), and it is 0.69 (0.71) for RR (on MC set). Also MC is better for 3 out of 5 tasks, worse for one and similar for another.

We find that the approach to correct codes is better than relaxing grammar. This implies that our code correcting algorithm does extrapolate the user's *intent*. However, it is unable to do so for 52% codes. We combine both the approaches to take each one's advantage. Here, we use MC approach for the correctable codes and RR for the rest. We henceforth call this the $RRMC$ approach. We use this for all further analysis.

Our second question is regarding the accuracy of our models. As reported before, the correlation between expert ratings is 0.72. We find $RRMC$ models have a mean correlation of 0.71 (Refer second last column of Table 3). Only for one task, the correlation is fairly low at 0.62. This shows that our approach can at most times provide human-competitive results.

Second, we also built models using compilable codes to predict their labels. We find that the accuracy of these models is much higher (on average, 0.85 as compared to 0.71 for UC). This is because we have a fairly informative feature, number of test-cases (TC) passed, for CC. This feature alone provides a correlation of 0.73 on average with expert ratings. Without this feature, the average correlation for CC falls to 0.70, comparable to 0.71 of UC. We do not have the benefit of the test case feature for uncompilable codes.[8] Even when we correct the codes, we do not semantically correct them to result in an informative TC feature.

Lastly, we wanted to analyse whether we could reuse the models trained on compilable codes (CC models) for uncompilable codes. Here, we use $RRMC$ approach to derive features and use the machine learning model learned on compilable codes. This way we do not require additional labelled UC for building a model. The first column in Table 3 shows the results of using the model trained on compilable codes directly. We find that that though the correlation is a little poorer (0.67 compared to 0.71), the MAE is significantly higher (0.81 compared to 0.50).

To improve the model, we used two insights. First, we drop TC as a feature in training our models on CC. As discussed above, the feature isn't predictive of grades for UC in the same way it is for compilable codes. Second, a comparable correlation and high MAE signals a systematic error. To test this hypothesis, we simply modify the distribution of the model output to match with that of the uncompilable code ratings (on the train set, explained in more detailed later). The results in Table 3 show that by fixing the distribution, there is a dramatic improvement in the MAE — it comes down to 0.52, comparable to 0.50 of model learnt on uncompilable codes. On removing the test case feature, we get a further improvement. The mean r becomes 0.71 and MAE 0.49, almost same as models learned on uncompilable codes.

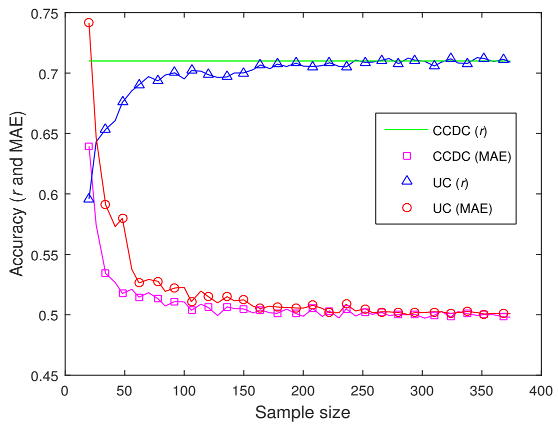

Figure 2: Accuracy of models built on a sample of compilable codes with distribution correction (CCDC) and uncompilable codes (UC). The x-axis denotes the sample size used and y-axis denotes accuracy in terms of r and MAE.

Distribution correction (DC): The grades predicted by

[8] The value of the TC feature is 0 for codes that we are unable to make compilable. We use the number of test cases passed by the compilable program for codes corrected by the MC approach.

Table 4: Distribution of PA scores for candidates with compilable and uncompilable programs. The table also includes the number of candidates selected for interview and the number of candidates who were hired.

	Rubric Definition	Compilable	Uncompilable
1	Code unrelated to given task	3361	1979
2	Appropriate keywords and tokens are present	3264	2125
3	Right control structure exists with missing data dependency	2547	1440
4	Correct with inadvertent errors	2955	1017
5	Completely correct	3828	-
≥ 3	Selected for interview	9330	2457
-	Hired	2986	565

Table 5: Examples of candidate submissions with uncompiling codes who got hired.

Program 1	Program 2
Input: String **Expected output:** String without vowels	**Input:** String **Expected output:** 1 if the brackets are balanced, otherwise 0

Program 1

```
1  class Solution
2  {
3  public static String vowel(String st)
4  {
5        char strChar[]=st.toCharArray();
6        String rev="";
7        for(int i=0;i<st.length();i++){
8     if(st.charAt(i)=="a" || st.charAt(i)=="A"){
9        rev=rev+"";
10    }else if(st.charAt(i)=="e" || st.charAt(i)=="E"){
11       rev=rev+"";
12    }else if(st.charAt(i)=="i" || st.charAt(i)=="I"){
13       rev=rev+"";
14    }else if(st.charAt(i)=="o" || st.charAt(i)=="O"){
15       rev=rev+"";
16    }else if(st.charAt(i)=="u" || st.charAt(i)=="U"){
17       rev=rev+"";
18    }else{
19       rev=rev+st.charAt(i);
20    }
21    return rev;
22    }
23 //Missing return statement
24 }
25 }
26
27
28
```

Program 2

```
1  public class Parentheses
2  {
3    public static int brackets(String str)
4    {
5      Stack<char> s=new Stack<char>();
6      for(int i=0;i<str.length();i++){
7        char st = str.charAt(i);
8        if(st=='('||st=='{'||st=='[')
9          s.push(str.charAt(i));
10       if(s.empty()){
11         s.push(str.charAt(i));
12         continue;
13       } else{
14         char cur = s.top();
15         if(str==')' && cur == '(') {
16           s.pop();
17         } else if(st=='}' && cur == '{') {
18           s.pop();
19         }else if(st==']' && cur == '[') {
20           s.pop();
21         }
22       }
23     }
24     if(!s.empty())
25       return 0;
26     return 1;
27   }
28 }
```

Program 1	Program 2
Compiler Errors **Lines 8, 10, 12, 14, 16:** Incomparable types *char* and *String* **Line 23:** Missing return statement	**Compiler Errors** **Line 5:** Unexpected type *char* **Line 16:** Undefined symbol *top*.
Actual corrections to be made **Lines 8, 10, 12, 14, 16:** Replace ” with ’ **Line 23:** Include *return* statement in the main block	**Actual corrections to be made** **Line 5:** Replace *char* with *Character* **Line 16:** Replace *top*() with *peek*()

CC models are transformed such that their distribution match the expert grade distribution of the UC. We do an equipercentile transformation on the train set of the UC.[9] One may note that this actually requires expert grades on the UC. Our hypothesis was that distribution matching would require a much smaller sample of labelled codes as compared to building a fresh model. To confirm this, we do a simulation: we bootstrap different sample sizes of labelled uncompilable codes and test the mean r/MAE for the distribution correction approach (CCDC) vs. building fresh models (UC). The results are shown in Figure 2.

We find that the MAE for the DC approach as a function of the sample size reduces much faster as compared to training fresh models. The DC approach has the right r throughout. To attain stability within 1% of MAE and r of the asymptotic accuracy, we require around 140 samples for the DC approach and 220 samples for building fresh models. This mean a reduction of about 33% in labelling effort. This becomes significantly large for creating models for multiple programming languages and for multiple raters. For 10 languages and 3 raters, it would lead to rating 2400 additional codes.

6 Development and Deployment

Our current system grades uncompilable programs written in C, C++ and Java. Our test delivery platform, scoring and feedback engine were developed in Python. It took us close to 15 months in designing and deploying the system in production for all the three languages. We can easily add new programming problems for the existing languages in the system. To scale the system to a new language, we need to develop the *feature extraction* component and train new machine learning models for it. With two developers, it takes around two months to add a new language to the system.

Case Studies

We deployed the **RRMC** model in the scoring engine of *Automata*, our online programming evaluation platform. A multinational technology company in China with more than $50,000$ employees used this product for their 2017-2018 campus hiring cycle for entry level software engineers. The company wished to find candidates who can think of correct algorithmic approach to solve a given programming problem. In the Automata report, we provide scores on Programming Ability (PA), Programming Practices (PP), Time Complexity (TC) and a total score. Earlier, uncompilable codes had a -1 score on Programming Ability. With incorporation of the new models, now uncompilable codes got a PA score based on our standard rubric. The total score is a combination of PA, PP and TC scores. A penalty for non-compilation was introduced in it. This helped score a compilable code higher than uncompilable codes, if they had the same score on all metrics.

[9]The equipercentile method is commonly used in test equating. Here, scores with the same percentile on the two distributions are considered equivalent. Scores from one distribution are mapped to the *equipercentile* score on the other distribution. Further details may be found in (Braun H. I. 1982; Kolen M. J. 1995).

All candidates first took the test. The company shortlisted candidates with a PA score of 3 or above for interviews. All interviewers were provided with the candidate report, which provided various test scores, the candidate programs and whether the programs were compilable or not. The interviewers were instructed to do an independent evaluation of the coding skills of the candidates. They asked candidates to work out the algorithmic solution for two programming problems during the interview. Based on how the candidate solves these problems, the final hiring decision was made. The recruiters made the decision by their own evaluation, together with transparent input on compile/non-compile from test.

A total of $29,600$ candidates took the test. Out of these 54% had a compilable code, 24% had a blank code and 22% had an uncompilable code. The PA score for candidates with compilable and uncompilable codes is shown in Table 4. The company used a cut-off of 3 or above on the PA score to interview candidates. We find that a total of 2457 additional candidates were selected for interview (an addition of 26% candidates). After the independent evaluation of interviewers, 565 selected candidates had uncompiling codes (an addition of 19% hired candidates). The company was able to hire much more efficiently. Also, many worthy candidates who would have got missed out by traditional program grading systems were hired.

In another study, a large IT services company in India used the system for hiring software engineers from campus. The company hires tens of thousands of software engineers every year and cite inability to fulfill all open positions as a major challenge. Using our tool, the company was able to improve shortlisting and selection rate by 30%-40%. They used an independent programming interview round to make hiring decisions. The increase in hiring rate led to big savings in sourcing cost, business cost of unfilled positions and greatly reduced time to hire.

We eyeballed some of the uncompilable codes of hired candidates in these studies. We show couple of examples in Table 5. One observes that the codes' logic is correct, but they do not compile due to incorrect declaration or use of data types. Another error is wrong placement of $return$ with respect to parentheses. We find that candidates with near-correct codes (semantically) makes compilation errors due to lack of knowledge of using the language. They also sometimes are unable to debug simply silly errors. One reason for this could be that compiler generated error messages aren't instructive enough to correct the errors (see 'Compiler Errors' in example). For instance, in (Becker et al. 2018), authors report that 64% students find multiple compiler error messages confusing and hard to debug.

7 Conclusion

In this work, we present the first approach to grade and provide semantic feedback on uncompilable codes. We use machine learning to find the correctness of the algorithmic approach of a program irrespective of its compliance to syntax. We require the AST and ST of the programs to derive informative features from them. We propose two methods to do this. The first method makes the code compilable by doing corrections. The second relaxes compilation rules to enable parsing of uncompilable codes. Each of these methods has its own advantages. The first provides a better judgment of coder's *intent*, while the second is able to parse a much larger number of programs. We combine both these methods in our final algorithm.

We grade uncompilable codes according to a rubric based on features derived from the ST and AST. We perform supervised learning on a sample of expert graded codes. We find that our machine learning models for uncompilable codes rival the internal consistency of experts. We further find that the models developed for compilable codes can be reused for uncompilable codes after normalization. This reduces the need of new labelled data on uncompilable codes.

To the best of our knowledge, this is the first work which attempts to separately treat the skill to propose the right algorithmic approach for a problem vs. being able to write syntactically correct code. This enables companies and evaluators to identify good coders even if their submitted programs do not compile. We identified 19% such coders in the case study with a company. The approach also enables coders to get semantic feedback on uncompiling codes.

In future, the holy grail is how we can further relax the need for writing a code following syntax and grammar rules — be able to grade pseudo-code or that written in natural language. This constitutes a specialized problem in the domain of natural language understanding. Our immediate area of interest is to investigate how to provide more fine-grained feedback for uncompilable codes.

References

Appel, A. W. 1997. *Modern Compiler Implementation in ML: Basic Techniques*. New York, NY, USA: Cambridge University Press.

Aspiring Minds. 2012. Automata. Accessed: 2018-09-02.

Becker, B. A.; Murray, C.; Tao, T.; Song, C.; McCartney, R.; and Sanders, K. 2018. Fix the first, ignore the rest: Dealing with multiple compiler error messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, 634–639. New York, NY, USA: ACM.

Bhatia, S., and Singh, R. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks.

Braun H. I., H. P. W. 1982. Observed-score testing equating: A mathematical analysis of some ets equating procedures. *Holland, Paul W.; Rubin, Donald B. (eds.) Test Equating. New York: Academic Press, 1982, p9-49* 267–288.

Gupta, R.; Pal, S.; Kanade, A.; and Shevade, S. K. 2017. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 1345–1351.

Head, A.; Glassman, E.; Soares, G.; Suzuki, R.; Figueredo, L.; D'Antoni, L.; and Hartmann, B. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, 89–98. New York, NY, USA: ACM.

Kolen M. J., B. R. L. 1995. *Test Equating, Scaling, and Linking*. New York: Springer-Verlag.

Le Goues, C.; Nguyen, T.; Forrest, S.; and Weimer, W. 2012. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.* 38(1):54–72.

Long, F., and Rinard, M. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, 298–312. New York, NY, USA: ACM.

Mechtaev, S.; Yi, J.; and Roychoudhury, A. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 691–701. New York, NY, USA: ACM.

Singh, R.; Gulwani, S.; and Solar-Lezama, A. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 15–26. New York, NY, USA: ACM.

Singh, G.; Srikant, S.; and Aggarwal, V. 2016. Question independent grading using machine learning: The case of computer program grading. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, 263–272. New York, NY, USA: ACM.

Srikant, S., and Aggarwal, V. 2014. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, 1887–1896. New York, NY, USA: ACM.

Tibshirani, R. 1994. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B* 58:267–288.