Clustering Code by Intent

13 June 2018

Annie Hu
anniehu@ stanford.edu

Elliott Chartock
elboy@ stanford.edu

Abstract

As enrollment in Computer Science courses skyrockets, a key question is how to maintain quality and consistency of grading at scale. Existing approaches to computer-assisted grading mainly focus on analyzing structure of compiled code. We introduce an NLP model for analyzing and clustering code snippets based on student approach, or intent, that considers both structure and semantic meaning. We find that the best performing models use well-chosen feature extractors, high-dimensional GloVe embeddings, and a deep neural network architecture, achieving best R^2 score of 0.5 in predicting normalized grade.

1 Introduction

Each year, hundreds of aspiring Computer Science students enroll in introductory level coding courses. Between assignments and exams, the teaching staff is inevitably left with the arduous and painful task of manually grading thousands of code snippets, tediously reading line by line to first understand the student's method and then assign a grade. Recent efforts to digitize the exam process have paved a path for more efficient grading techniques. Student code is sprinkled with useful plain-text indicators, such as comments, variable names, and function names, that can be used to identify student intent. We propose a method for clustering code snippets to facilitate grading.

In Section 2, we discuss related work and a high-level overview of how our project integrates and extends previous projects. In Sections 3 and 4, we examine the dataset and useful featurizers. In Section 5 we outline our technical approach, which incorporates pre-trained word embeddings and deep neural networks. Finally, in Sections 6 and 7 we review our results, visualize learned clusterings, and in Section 8 we conclude with our project learnings and directions for future work.

2 Related Work

Most existing work on clustering code snippets examines their syntactical similarity through constructing Abstract Syntax Trees (ASTs) or otherwise measuring program similarity solely based on token patterns and code structure (Yan et al., 2018). However, we believe that grading is most efficient when we can cluster code based on *intent*. In other words, given many possible paths to a working solution, we want to be able to cluster intermediate solutions based on which path they are following. This is especially relevant for exam grading as most solutions are not in final, compilable forms.

(Hindle et al., 2012) provided a breakthrough insight through their work modeling software using statistical natural language models. An "essential fact" underlying the field of natural language processing is that though natural language is rich, complex, and powerful, most human utterances are largely regular and predictable. They showed that the same applies to software, or that though programming languages are also complex and powerful, programs written by humans are repetitive and predictable in a similar way to natural language, and thus can be modeled using similar techniques. Notably, they removed comments from code before extracting n-grams, but we believe student comments in exam solutions will be key indicators of approach.

Since there is not much existing work on clustering code based on *intent* or NLP approaches, we look to the related problems of clustering queries or documents based on intent for potential approaches. Within this field, (Cheung and Li, 2012) implemented an unsupervised query clustering model by extracting semantic features from Freebase entities and domain-independent lexical features from a Wall Street Journal corpus. Impressively, they were able to achieve over 90% accuracy as judged by Amazon Mechanical Turk.

Thus, our goal is to combine structure-based and semantic-based approaches, where word embeddings can be used in conjuction with word order and program structure, to produce useful representations and clusterings of code – an exciting, not-much-explored field!

3 Data

We use a dataset comprised of exams of Stanford CS106A students collected through *BlueBook*, a "lightweight, cross-platform, computer-based, open source examination environment that students can run on their laptops." (Piech and Gregg, 2018). Because grading rubrics are usually designed with different point deductions for different approaches, grading is both more efficient and more consistent when solutions with similar approaches are graded together. Chris Piech has generously offered anonymized *BlueBook* data from three quarters of 106A, encompassing over 7,000 exam problems with corresponding student answers, scores, and grading criteria, with the goal of eventually incorporating a clustering model to improve grading.

For training and testing, we use a "held-out" method where our validation set consists of all submissions for one problem, our test set all submissions for a separate problem, and our train set all other submissions – this creates 6618 train, 362 validation, and 362 test problems. This held-out method simulates how our model would be used in a real grading setting, as it would need to cluster submissions for a totally unseen problem.

4 Data Cleaning

Our inputs in this problem are unstructured raw text of student code written and submitted during an exam. To account for student spelling errors, common Java naming conventions, and inherent code snippet redundancy, we apply a series of preprocessing and featurization techniques to transform the raw text into a feature vector that is input into our model.

4.1 Preprocessing

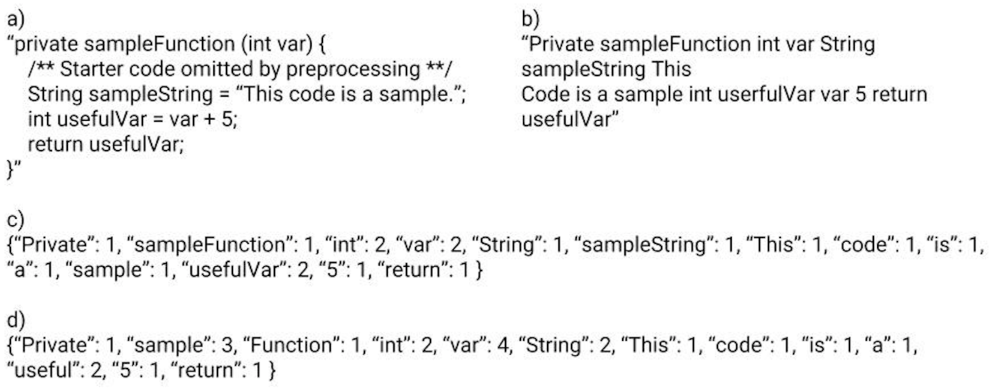
Each training sample is a student code snippet that is written in Java syntax. We clean the data by omitting all white space, punctuation, and any starter code that is identical in all submissions. We leave the text in its original case form so that we can build feature vectors that account for camel case variable names, a common naming principle in Java. Figure 1 shows a sample student code snippet along with corresponding preprocessed and featurized representations (discussed in Section 4.3)

4.2 Bag-of-words

As a simple baseline featurizer, we apply the bag-of-words model to the preprocessed code snippets. The bag-of-words model is a method for representing text as a multiset of its words. With this approach, order and semantic meaning are not considered, only the occurrence count of each word in the text. We train a linear regression model on the bag-of-words representation of each training sample to test if simple word counts correlate with a code snippet's correctness. This is a very naive first pass that is used both as a sanity check for our training pipeline and as inspiration for future more complex featurizers. We visualize the bag-of-words model in Figure 1c.

4.3 Camel case separated Bag-of-words

We build on the vanilla bag-of-words model with a camel case separated bag-of-words-model. This featurizer first separates camel case variable names and then applies bag-of-words. Consider a code snippet with a function named '*getCharacterCount*' and a local variable named '*characterCount*'. By the vanilla bag-of-words model, we will train our models on features that don't capture any semantic overlap in the two student-chosen variable names. However, with the camel case separated bag-of-words model we will now pass into our model an input feature that captures the duplicate '*character*' and '*count*' naming choices. While it is not clear that this method will offer any benefits over the vanilla bag-of-words model, this is an important next step towards building a



a)
```
"private sampleFunction (int var) {
    /** Starter code omitted by preprocessing **/
    String sampleString = "This code is a sample.";
    int usefulVar = var + 5;
    return usefulVar;
}"
```

b)
"Private sampleFunction int var String sampleString This Code is a sample int userfulVar var 5 return usefulVar"

c)
{"Private": 1, "sampleFunction": 1, "int": 2, "var": 2, "String": 1, "sampleString": 1, "This": 1, "code": 1, "is": 1, "a": 1, "sample": 1, "usefulVar": 2, "5": 1, "return": 1 }

d)
{"Private": 1, "sample": 3, "Function": 1, "int": 2, "var": 4, "String": 2, "This": 1, "code": 1, "is": 1, "a": 1, "useful": 2, "5": 1, "return": 1 }

Figure 1: a) Sample student code snippet b) Preprocessed sample code snippet c) Bag-of-words representation d) Camel case separated bag-of-words

featurizer that uses word embeddings. Continuing with the above example, any pretrained word embedding map will consider the word '*characterCount*' as an '*UNK*' character, whereas by separating the word by the camel case mechanism, we can capture the semantics of the words '*character*' and '*count*' intended in the student's variable name choice. We visualize the bag-of-words model in Figure 1d.

4.4 Word Embeddings

We featurize student code using word embeddings that encode semantic meaning. GloVe is a popular embedding technique in which an embedding matrix is learned by applying dimensionality reduction to a context co-occurrence matrix (Pennington et al., 2014). With X_{ij} as the number of times word X_i occurs in context X_j, word vectors $w, \tilde{w} \in \mathbb{R}^d$, and bias vectors $b, \tilde{b} \in \mathbb{R}^d$, Glove minimizes the following reconstruction loss in order to find the best low dimensional representation:

$$\textbf{Loss} = \sum_{i,j=1}^{V} f\left(X_{i,j}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

Our hope is that using such embeddings will allow variable names and comments to provide useful signals to the model. For the LSTM model we concatenate the GloVe embeddings of the processed words to format the feature input as a time-series $X \in \mathbb{R}^{T \times d}$, for T words in the code snippet and GloVe embedding size d. For non-recurrent models we experiment with two GloVe collation featurizers. We produce the feature input $X \in \mathbb{R}^d$ by 1) summing the GloVe embeddings of the processed words or 2) averaging the GloVe embeddings.

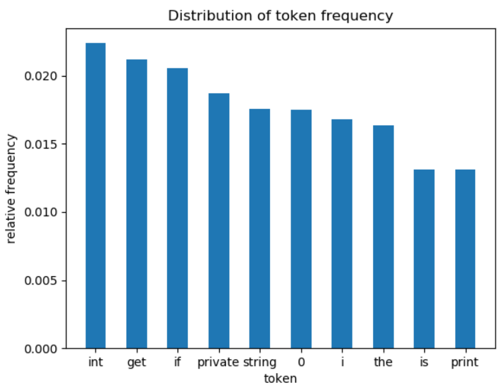


Figure 2: Distribution of relative token frequency over all submissions. Of these most frequent 10 tokens, 5 are Java keywords ("int", "if", "private", "string", "print").

4.5 Keyword filtering

Most of the tokens in code submissions are code keywords – for Java, these include tokens like "int", "while", and "void". Figure 2 shows that 5 out of the 10 most frequent tokens in our dataset are such keywords. We would like to investigate how important code keywords are versus words with semantic meaning in clustering code snippets. Thus one of our feature extractors filters code keywords, with the option to consider only keywords, only non-keywords, or both.

4.6 Spellchecking

Since we are using raw exam data, we find that many submissions contain misspelled words, which interfere with the effectiveness of word embeddings or learned vocabularies. We use a python

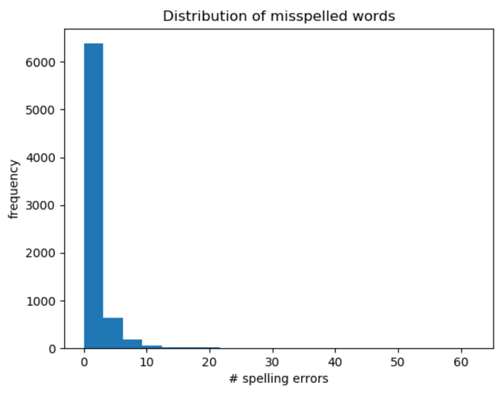


Figure 3: Distribution of misspelled words over all answers, where we count a word as misspelled if it contains all ASCII characters, is not a code keyword, and is not found in our GloVe lookup. Notice that while the distribution is strongly skewed left, around 500 submissions had between 5-10 misspellings, and some outlier submissions had over 40 (which may have been variable names such as "answ").

spellchecker[1] to correct these misspellings in all submissions, being careful to preserve Java keywords and variable names such as "experiment1". Figure 3 shows the distribution of proportion of misspelled words across all submissions.

5 Methods

5.1 Regression Task

We first approach this problem as a regression task. Given the input vector of a student code snippet, we try to predict the normalized score that the submission will receive, where the normalized score is calculated by dividing the student score by the total possible points for that question. We predict normalized score because absolute score is arbitrary and often meaningless; the same question could be worth 1 point or 10 points, according to the teacher's discretion. We build a baseline linear regression model, and then aim to outperform these models with deeper and more complex models as described in Sections 5.3 and 5.4.

The loss function for this task is the mean squared error (MSE), which measures the average of the squared difference between our predictions and the ground truth values.

$$\mathbf{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

5.2 Classification Task

We then approach this problem as a binary classification task in which positive and negative labels correspond to the student code being fully correct or having some error, respectively. This classification task is one form of guiding intelligent clusters that will facilitate grading. We build a baseline logistic regression model, and then improve our results using our neural network models.

We use binary cross entropy (BCE) loss to train our models. BCE loss can be computed as in (1), which cleverly combines a piecewise function into a single, differentiable loss function.

$$\mathbf{BCE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} -[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \quad (1)$$

5.3 Fully Connected Neural Network

To match the complexity of the task, we implement a fully connected neural network with two affine layers. Formally, for inputs x, this network performs

$$h_1 = max(W_1 x + b_1, 0)$$

$$out = W_2 h_1 + b_2$$

, where W_1, W_2, b_1, b_2 are learnable model parameters and *out* is the single float predicted. For the classification task, we apply an additional

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

transformation to squash this predicted value to a probability value between 0 and 1.

5.4 Long Short Term Memory Network (LSTM)

To better capture the structure of each solution, we also experiment with Recurrent Neural Network (RNN) architectures, which are known for their usefulness in modeling sequential data. An LSTM is a type of RNN that is especially powerful in modeling longer sequences.

[1]https://github.com/barrust/pyspellchecker

Table 1: Regression results

Model	Best Regression R^2	Best Classification F_1	Average R^2	Average F_1
Linear Regression BOW	-170	-	-6150	-
Logistic Regression BOW	-	**0.94**	-	0.63
Neural Network BOW	0.42	0.42	-1.17	0.12
Neural Network Sum GloVe	**0.50**	0.58	-0.22	0.17
Neural Network Avg GloVe	0.41	0.49	-0.13	0.14
LSTM GloVe	0.06	0.11	-0.29	0.02

Given a sequence of vectors $x_1, ..., x_T$, a Vanilla LSTM with L layers computes, at a single timestep t and layer l,

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g, \quad h_t^l = o \odot \tanh(c_t^l)$$

$$y_t^l = W_{hy}^l h_t^l$$

where $\odot$ indicates element-wise multiplication, i, f, o, g, c_t^l are the H dimensional input gate, forget gate, output gate, block gate, and context vector respectively at the tth timestep and lth layer. We train our models with $L = 2$ and $H = 100$. All model outputs are ignored except y_T^L. In the regression setting, we perform MSE loss on y_T^L, the model prediction. For the classification task, we apply a final sigmoid to y_T^L to reinterpret the output as a probability rather than a normalized score prediction, and backpropagate using BCE loss.

6 Results

With tuning, our final models used the following hyperparameters:

- **GloVe dimension:** 200
- **Batch size:** 64
- **Epochs:** 30
- **Learning rate:** 0.001; 0.01 (LSTM)
- **Momentum:** 0.5

6.1 Evaluation Metrics

We evaluate our regression problem using the coefficient of determination R^2, a standard statistical measure of what proportion of the variance in our predictions can be explained by variance in our input features. With truth values y_i, predicted values $\hat{y}_i$, and mean of truth values $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, this is calculated as

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}.$$

The best possible R^2 value is 1.0 (indicating perfect explanation of predictions by input features), and smaller or negative values indicate worse model performance.

The classification task is evaluated using the F_1 score. As opposed to accuracy, F_1 accounts for the imbalance in positive and negative labels by taking the harmonic mean of precision and recall. In other words, if 80% of the dataset has a negative label, a model that ignores the input and always predicts negative will yield a high accuracy but a low F_1 score.

6.2 Experimental Results

We trained a separate model for each held-out exam question, recording the best and average metrics across all questions. Table 1 shows the performance of our implemented models. Even using a simple Bag-of-Words model, our neural network significantly outperformed a linear regression baseline. As expected, incorporating GloVe embeddings also improved model performance, as the model was able to recognize semantic differences between variable names, method names, and comments. We found that summing GloVe vectors during feature extraction performed better than averaging vectors, and hypothesize that incorporating both may improve performance further. Our LSTM performed much poorer than expected, especially since we believed capturing sequential structure of code would be important in effectively determining intent. This may be due to our relatively small dataset – we believe training a larger dataset for more epochs, and possibly with more parameter tuning, would better leverage the power of this more complex model.

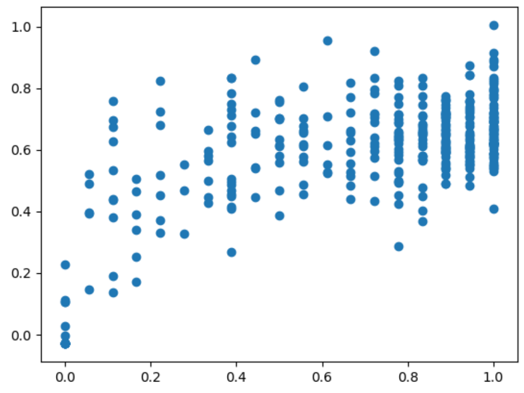


Figure 4: Scatterplot of predicted scores (y-axis) vs. true scores (x-axis) from our Neural Network using summed GloVe embeddings.

6.3 t-SNE Visualization

Unsupervised clustering is difficult primarily because we lack a clear-cut evaluation method as our data does not have any labeled clusters. We considered these regression and classification problems because we believed that a model that could perform well on those tasks would create natural clusterings in the data – in other words, the human grades could serve as a proxy for clustering via intent.

Other unsupervised clustering approaches (Gross et al., 2012) have used t-SNE visualizations of both predicted clusters and metrics such as intra-cluster radius and inter-cluster separation to evaluate their model. Such visualizations will be useful for us to visualize our model errors, and may also prove to be a good measure of how well our model is performing.

A favorable property of t-SNE is that the algorithm preserves relative distance between two vectors. We apply t-SNE two-dimensional reduction to the learned 100-dimensional embeddings of the student code snippet and visualize the test set in Figure 5. A good code snippet embedding, as in the right diagram, should yield visibly separable clusters of negative (blue) and positive (red) labels.

7 Analysis

We found that our model performed better when as many semantic features were extracted as possible. Thus, separating camel-case and snake-case variables, adding word embeddings, using higher-dimensional word embeddings, and adding spellchecking greatly improved our ability to generalize to unseen code problems. We also ran an experiment using only code keywords as features. This performed extremely poorly, whereas an experiment using only non-keywords as features performed not significantly worse than models that did not filter based on keywords at all. This shows that semantic features are very important to our model.

Many of the code snippets that both our regression and classification models performed poorly on seemed like examples that an LSTM would be ideally suited to handle. For example, submissions with comments such as “i dont have time” or with excessive commenting should be clear indicators (to a human grader) that the submission is likely incomplete. However, our model is not consistently able to identify these as low-performing submissions. Since we use pretrained GloVe embeddings, we are also limited by the tokens found in GloVe. Many common tokens in code, such as “hash,” “arraylist,” and “str,” are not everyday words that would have useful pretrained embeddings, and we are currently ignoring these tokens in our model. We could glean even more information from each submission by including these code-specific tokens by building our own vocabulary or training our own embeddings. By training our own embeddings, we could even perhaps learn common code structures such as the syntax around setting up a loop, which should improve model performance.

Ultimately, we believe we were limited by the small size of our training set. With only 7,000

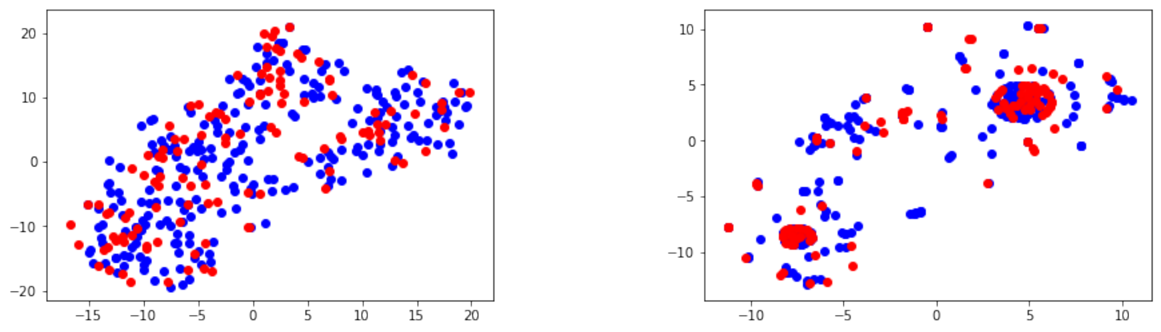


Figure 5: t-SNE visualizations of a more difficult code problem (left) and a simpler code problem (right), where red represents a perfect solution and blue represents imperfect. The simpler problem shows clear clusters, which corresponds with its high F1 score of 0.94. The more difficult problem shows less clear separation of clusters, and especially less separation between perfect and imperfect solutions, which corresponds with its lower F1 score of 0.58.

submissions, it is difficult to generalize to a completely unseen code problem. The model tended to perform better on simpler problems, or problems with a more rigidly defined structure – while we achieve promising results here, especially with a Logistic Regression model, these types of problems fit less within our goal of clustering based on intent. The success of our simple Logistic Regression model compared to more complex models additionally suggests that more data would be beneficial to avoiding overfitting and ensuring that the greater complexity of a Neural Network or LSTM can be effectively leveraged.

8 Conclusion / Future Work

We showed that code contains important semantic features that allow us to make meaningful predictions of code quality, and by proxy code intent. We implemented several feature extractors and preprocessing steps, including separation of variable and method names, stripping starter code, accounting for Java keywords, fixing spelling errors, and using pretrained GloVe embeddings. We found that a Fully Connected Neural Network with summed GloVe embeddings performed best on predicting a normalized score, and a Logistic Regression model performed best on predicting whether a submission was perfect.

The most important step in future work is directly learning our own token embeddings, so we can properly handle tokens that are unseen in GloVe, learn common code structures such as loops, and create a better encoding of an entire submission for clustering. Our dataset also contains “rubric items,” or a list of predetermined potential errors. These ultimately proved too messy in our dataset, as each exam problem used a different rubric-naming system, but if these were further preprocessed or stored more consistently in future datasets, they would be a much better proxy for code intent than final score.

9 Acknowledgements

We are so grateful for Chris Piech in allowing us to use BlueBook data and providing guidance throughout the project, as well as the CS224U teaching team for their feedback and support.

References

Jackie Chi Kit Cheung and Xiao Li. 2012. Sequence clustering and labeling for unsupervised query intent discovery. In *Proceedings of the fifth ACM international conference on Web search and data mining*.

Sebastian Gross, Bassam Mokbel, Barbara Hammer, and Niels Pinkwart. 2012. Feedback provision strategies in intelligent tutoring systems based on clustered solution spaces.

Abram Hindle, Earl Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 837–847. ACM Digital Library.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Chris Piech and Chris Gregg. 2018. Bluebook: A computerized replacement for paper tests in computer science. In *Proceedings of the 49th ACM Technical*

Symposium on Computer Science Education, pages 562–567. ACM Digital Library.

Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. Tmoss: Using intermediate assignment work to understand excessive collaboration in large classes. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 110–115. ACM.