CS 698 PROJECT REPORT

SUPERVISED CLASSIFICATION OF COMPETITIVE PROGRAMMING QUESTIONS

April 16, 2017

Nimesh Ghelani
Student ID: 20663356
University of Waterloo

INTRODUCTION

"Competitive programming is a mind sport ...", "involving participants trying to program according to provided specifications"[1]. Competitive programmers solve questions by writing code which is expected to run correctly on a set of secret test cases. The solutions also needs to be fast and must complete within a specified amount of time. Thus, identifying an efficient set of algorithms to solve a given problem is crucial.

While the questions in such competitions are related to algorithms, data structures and math, they are usually classified under narrow topics such as *combinatorics*, *dynamic programming*, *depth first search*, *max flow* and so on. Various online platforms exist where users can practice on a variety of questions. Despite the popularity, only recently have these platforms started to tag their questions with topics (manually by the users or problem setters). This tagging process is still completely user centric, as a result of which, there still remain a large number of old questions without any relevant tags.

The aim of this project is to use these user-tagged problems to predict the tags/topics of untagged problems. In other words, we aim to build a supervised classifier which can auto label problem with their respective topics. There haven't been many previous attempts to tackle this specific problem. We believe that this is because there is no shortage of newly tagged questions due to which there isn't any motivation to tag older questions. Thus, there isn't a baseline we can compare our results with.

There are three data points relating to a given programming question; problem statement, editorial and submitted solution. While the problem statements are highly convoluted and cryptic to conclude anything about the question topic (this is the part which competitive programmers do while solving a questions), editorials are not widely available specially for older problems. This leaves us with the submitted solutions, more specifically the source code of solutions which were correct.

Source codes in the competitive programming setting usually have common patterns and cues which are subject to the type of algorithm or data structure being used. For example, many solutions using *dynamic programming* contain array names like **dp** (which is a short notation for *dynamic programming*). Solutions using *disjoint set unions* often have blocks of code which follow a common flow. An experienced competitive programmer can glance over a source code and quickly pick up these patterns and cues to determine the type of algorithm used in the solution. This project attempts to effectively learn these patterns present in the source code.

[1]`https://en.wikipedia.org/wiki/Competitive_programming`

In this project, we train and evaluate three classification techniques: Convolutional Neural Networks, Multinomial Naive Bayes, and Logistic Regression. We also compare performance of different feature when using these classifiers.

RELATED WORK

Although there isn't much effort made towards this specific problem, a blogger[2] performed a simple experiment to predict if a given source code used *binary search* or not. The method uses a standard Convolutional Neural Network with two layers of convolutions (50x1 and 25x1) with max pooling (4x1) followed by a fully connected layer with 256 neurons and a softmax layer. This network achieves 78% accuracy on binary classification of source codes. However, the input to the network is a one dimensional vector of ascii values. The author mentioned this potential flaw and the possibility of improvement by replacing the input with a higher dimensional vector.

Source code classification can be viewed as a form a text classification. Many text classification approaches using CNN are based on Yoon Kim's work[1]. The CNN approach mentioned in the following sections is based on this work, in addition to the work mentioned in the previous paragraph.

In Software Engineering, a related problem of interest is *type 4 code cloning*[2]. It is defined as *"Two or more code fragments that perform the same computation but are implemented by different syntactic variants"*. However, the definition of *type 4 code cloning* is too strict for our use case. Moreover, much of the research is present for *type 1-3 code cloning*. On a related note, Ugurel et al[3] developed an SVM based approach to classify software repositories into topics like *Database*, *Games*, and so on. A similar problem was addressed by Mou et al[4] as part of their work in using CNN with AST (Abstract Syntax Tree) representation of source code.

While CNNs reduce the need for feature engineering, Logistic Regression and Naive Bayes heavily rely on the right set of features. Engineering features from source code can be a daunting process unlike simple text documents. Byte levels n-grams have been used effectively as features for problems like spam filtering[5]. Byte 4-grams are simple features which can easily fit in the memory (2^{32} maximum features).

[2] `https://blog.anudeep2011.com/machine-learning-everywhere-why-not-in-competitive-programming/`

DATASET AND EVALUATION

We gather source code data from an online competition platform called Codeforces[3]. We crawled correct C/C++ solutions for problems tagged with one of the four topics: *dp* (Dynamic Programming), *binary search*, *dsu* (Disjoint Set Union), and *graphs*. Since a question can be tagged with multiple topics, we excluded problems which were tagged with more than one of the above four topics. To reduce noise, only the problems with at least 25 correct solutions were selected.

Text cleaning was performed on the source code. Text cleaning is beneficial since it helps to capture most important parts of the source code when feeding it to a CNN (where the input vector size is fixed and relatively smaller than length of an average source code). Cleaning process involves removal of consecutive whitespace, preprocessor directives and blank lines. Source codes with length greater than 1000 characters are discarded (CNN input vector size is fixed at 800). The composition of the remaining data is summarized in Table 1.

	# Source Codes	**# Questions**
graphs	4605	77
dsu	2229	36
dp	6767	110
binary search	3757	65

Table 1: Dataset Summary

The models are evaluated by randomly splitting the dataset into training (80%) and test (20%) sets. A naive split on a set of source codes can cause overfitting since test and training sets may contain source codes which are solutions to the same problem. In order to avoid this, we perform our splits on the question set, making sure no two solutions in the training and test set share the same question.

We report on two accuracy measures: source code classification accuracy and problem classification accuracy. Source code classification accuracy is computed as

$$\frac{\#\, source\ codes\ correctly\ classified}{\#\, of\ source\ codes} \times 100$$

A class of each problem is decided by performing a majority voting of classes predicted for its solutions/source codes. Problem classification accuracy is then computed as

$$\frac{\#\, problems\ correctly\ classified}{\#\, of\ problems} \times 100$$

[3]`https://www.codeforces.com`

TECHNIQUES

Logistic Regression and Multinomial Naive Bayes

We use scikit-learn[4] to build our classifiers. Since we have a multiclass classification problem, we use *One vs Rest* strategy using multiple binary logistic regression classifiers. Multinomial Naive Bayes is inherently a multi class classifier.

We experiment with two set of features.

- The *alpha token* features are populated by ignoring all non-alphabet and non-whitespace characters and tokenizing the remaining text delimited by whitespace.
- The *byte 4-gram* features are populated by moving a window of 4 bytes across a given source code.

Both these features are then weighted using $TF \times IDF$ where *TF* (Term Frequency) of a feature in a document is the number of times a feature occurs in it, and *IDF* (Inverse Document Frequency) of a feature is $log\frac{N+1}{n+1} + 1$, where N is the total number of training examples, and n is the number of training examples containing this feature. The weights are then $l2$ normalized.

Convolutional Neural Networks

We use tensorflow[5] to build our CNN classifier. We use two convolution layers (patch size $x \times 25$ and 1×15 with stride of $1,1$) with max pooling (patch size 1×4 with stride of $1,4$), followed by a fully connected layer with 256 neurons and then a softmax layer with 4 output neurons. x is the width of the input tensor, which is dependent on our choice of input features. We restrict the size of our input sequence to 800 characters, Source code with size greater than 800 characters are clipped (from the beginning, due to author's intuition that less rich features are present in the beginning of the solutions).

We experiment with three set of features.

- *Simple character* strategy ($x = 1$) maps all visible characters (from ASCII 32 to 127) to a 800×1 input tensor with ascii values of corresponding characters. We train this CNN for 50 epochs.

[4]`http://scikit-learn.org/`
[5]`http://tensorflow.org/`

- *96d character* strategy ($x = 96$) maps all visible characters to a 800 × 96 input tensor. Due to resource constraints, we train this CNN for 25 epochs.
- *28d character* strategy ($x = 28$) maps all alphabets (converted to lower case), newline and space to a 800 × 28 input tensor. We train this CNN for 50 epochs.

Results and Discussion

	Source Code Classification %	**Problem Classification %**
cnn-simple	36.12	33.96
cnn-96d	53.64	62.26
cnn-28d	53.54	62.26
lr-alpha	61.34	69.81
lr-byte	**63.67**	75.47
nb-alpha	45.31	79.25
nb-byte	46.36	**86.79**

Table 2: Results

The results are summarized in Table 2.

All classifiers perform significantly better than random (25%). Surprisingly, Naive Bayes and Logistic Regression with simple features significantly outperform CNNs. An interesting contrast is between the source code and problem accuracy rate of Naive Bayes. This is probably due to the other classifiers performing very well for just one or two classes which have relatively higher number of solutions available per problem. An experiment with a larger and balanced training data should confirm the actual problem.

As expected, CNN with simple ascii weighted input features underperform compared to higher dimensional input features. An interesting result is how similar the CNN with *28d* input and *96d* input are. Training the latter for longer epochs should reveal a better contrast between these two strategies.

We also notice a significant improvement when using *byte 4-gram features* instead of *alpha tokens* in the Logistic Regression and Naive Bayes classifiers.

Since problem class is determined through majority voting of its solutions (or source codes), more data would directly improve problem classification accuracy. In our experiments we had an average of around 60 solutions per question. Most questions in online platforms have over 100s if not 1000s of correctly submitted solutions.

CONCLUSION AND FUTURE WORK

The results come as a surprise as simple classifiers perform better than complex CNNs in a fraction of training time. This validates the idea that CNNs (or neural networks in general) require intelligent hyperparameter tuning in order to perform well. The CNN model used in this project is far from the optimal and with right hyperparameters and computational resources, we believe that CNNs can perform much better.

The results also raises the question of simplicity vs complexity when selecting an appropriate model. The logistic regression and naive bayes classifier perform good enough to be practically used. They took less than a minute to train on our computer with very few lines of easy to understand code. For such problems, is it really worth to write complex deep neural network code which takes hours to train, with the added complexity of choosing the right hyper parameters?

This project lays down various directions for future work. Better feature dependent classifiers (Logistic Regression, Naive Bayes, etc) can be developed through sophisticated feature engineering techniques. Building a CNN which performs as good as the baseline set by this project will be an interesting challenge as well. We used modest amount of training data in our experiments, although there is a lot more available. Techniques using deep neural networks greatly benefit from more data. Incorporating more classes, programming languages (we only used C/C++) and multi label classification (Problems can have more than one topic) are some ideas which can lead to more practical applications of the techniques discussed in this paper.

Bibliography

[1] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[2] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[3] Secil Ugurel, Robert Krovetz, and C Lee Giles. What's the code?: automatic classification of source code archives. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638. ACM, 2002.

[4] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. *arXiv preprint arXiv:1409.5718*, 2014.

[5] Vlado Kešelj, Evangelos Milios, Andrew Tuttle, Singer Wang, and Roger Zhang. Daltrec 2005 spam track: Spam filtering using n-gram-based techniques.