Predicting algorithmic approach for programming problems from natural language problem description

Ashish Bora
ashish.bora@utexas.edu, UTEID : ab58952

Abhishek Sinha
as1992@cs.utexas.edu, UTEID : as76588

Abstract

In this paper, we study the problem of predicting the algorithmic approach useful for solving a programming problem given its natural language description. As an auxiliary task, we also try to predict the difficulty of a problem given its natural language description. In order to solve this problem, we built our own dataset using 2 popular algorithmic programming websites, namely, Codechef `www.codechef.com` *and Codeforces* `www.codeforces.com`*. In this paper we present an attempt at solving this problem.*

We use hashing, bag of words and word embeddings (word2vec) for feature extraction. We train a Long Short Term Memory (LSTM) network for the tasks. We also propose and evaluate a novel pre-training idea specific to our problem. The code for this project can be found at `https://github.com/AshishBora/nlp-project`*.*

1. Introduction

Most taks in natural language processing are highly obvious to humans. Consider for example, WSD, POS tagging, sentiment analysis, discourse analysis, etc.; any human with a reasonable language understanding can easily do these tasks. In fact most algorithms for these tasks are benchmarked against human performance. In contrast consider the following input text:

Alice loves simple strings! A string t is called simple if every pair of adjacent characters are distinct. For example ab, aba, abc are simple whereas aa, add are not simple. Alice is given a string s. She wants to change a minimum number of characters so that the string s becomes simple. Help her with this task.

Given a problem like the one above, a competitive computer scientist, after some deliberation, can recognize that this algorithmic task can be efficiently solved using *dynamic programming*.

In this paper we make an attempt at making a system which can do the same. i.e. We study the problem of predicting the algorithmic approach for solving an algorithmic problem given its natural language description. We emphasize that the problem to be solved is expressed more as a riddle rather than in a logical specification. In contrast to Natural Language Programming [7], in our task the algorithm is not directly told what is to be done, it has to infer that, albeit implicitly.

Predicting the algorithmic tag isn't a trivial task even for humans. Moreover for this task, it isn't the exact words used that matter as much as the deeper meaning of the word and the problem text. For example the subjects, situation, items exchanged, motives, and so on, can all be replaced while still keeping the underlying algorithmic problem largely unchanged. As a result, techniques which only look at surface statistics are unlikely to be effective. Thus, the main approach we tried was to use a Long Short Term Memory[2] (LSTM) on the sequence of word-level representation of the problem text. LSTMs are known to be Turing complete [5], i.e. given enough memory, time and processing power, they can compute everything that is computable. More recently, LSTM networks were successfully trained for natural language problems [6].

Most of the earlier work in this area was centered on applying NLP to solve simple algebra problems end to end ([1], [3]). These approaches took simple word problems as input and the goal was to parse and understand the problem, solve it and report the final answer. However, most of these problems were trivial enough that they could be solved by a human with high school math skills. On the other hand, the problems we are looking at are much more complicated. Hence, we have restricted our objective to predicting the solution category rather than actually solving the problem (like generating code or pseudo-code).

2. Problem Definition

2.1. Approach Prediction

In this problem, the input is a natural language question (as described in the introduction) and the desired output is its algorithmic category (such as dynamic programming).

2.2. Difficulty Prediction

In this problem, the input is a natural language question (as described in the introduction) and the desired output is its difficulty category.

3. Related Work

As described in the introduction section, most of the earlier work in this area was on using NLP to solve simple arithmetic word problems. In [1], the authors try to solve simple *Mathematical Word Problems*. They use Random Forest along with simple word and sentence level features to classify the problem into 3 arithmetic problem types (Join and Separate, Part-part-whole and Comparison). They also extract the numerical values in the problem text. Based on the inferred arithmetic problem type, they then apply the appropriate arithmetic transformation to generate the final answer.

In [3], the authors solve algebra word problems which are more complicated than [1] but these are still very simplistic as compared to the complexity of the problems we consider. The authors use a set of equation templates (which can be learned). They also computes sentence and word level features. To compute the most probable template assignment, the authors use a logistic regression model to score the problem template pair. The solution outputs, then, is the one computed using the most probable assignment.

In terms of methods, LSTMs have been used in the Natural Language Community for several problems such as language modeling [8], machine translation [6]. Word2Vec embeddings first described in [4] have also been found to be useful in a number of NLP tasks.

4. Datasets

4.1. www.codeforces.com

We scraped 2848 problems along with their tags from `www.codeforces.com/problemset`. There are multiple tags for each problem indicative of the algorithmic approaches useful for solving that problem. For example the problem above has tags: dp, hashing, strings, two pointers, greedy.

We observed that the first tag is the most informative and hence chose that as the tag to be predicted. Further we remove the tags which are very generic (like data structures, implementation, etc.). Finally we are left with 1954 problems and 30 different tags. The top 5 categories with their occurrence in percentage are – dp : 17.92, greedy : 15.87, binary search : 11.55 dfs and similar : 10.64, combinatorics : 06.37, geometry : 05.57. In the rest of the text, we will call this dataset, with single tags as 'codeforces'.

4.2. www.codechef.com

The data on this website is very different from the one above. Although, the website does provide some tags for algorithmic approaches for each problem, they are often mixed among a host of other tags like the day the problem was posted and problem submitter's name.

On the other hand the website neatly categorizes problems by their difficulty level. Further, counts of successful submissions for each problem and the overall accuracy for each problem are provided. Thus, the tags in this dataset are very complementary to the first one.

We were able to scrape 4898 problems with all the auxiliary information. The break-up of the dataset by level is as follows: school(very easy): 73, easy: 397, medium: 422, hard: 263, challenge (very hard): 90, extcontest: 3653. The 'extcontest' category contains problems from external contests. These problems are of various levels and thus, we do not know difficulty of individual problems. In the following, we will call the dataset with difficulty level as 'codechef-diff'.

5. Feature Extraction

From our literature survey, we found that for related tasks such as solving simple mathematical problems, removing stop-words, lemmatizing and stemming lead to comparatively worse performance. This suggests that mathematical word problems are very sensitive to minute changes in the text and we expect a similar pattern in our task as well. Thus, we decided not to any word or phrase level preprocessing. Instead we explore the following three approaches for feature extraction:

5.1. Hash Vectorizer

We use hashing vectorizer to convert a collection of text documents (stream of words) into a stream of hash values. We specify a maximum vocabulary size and only hash the

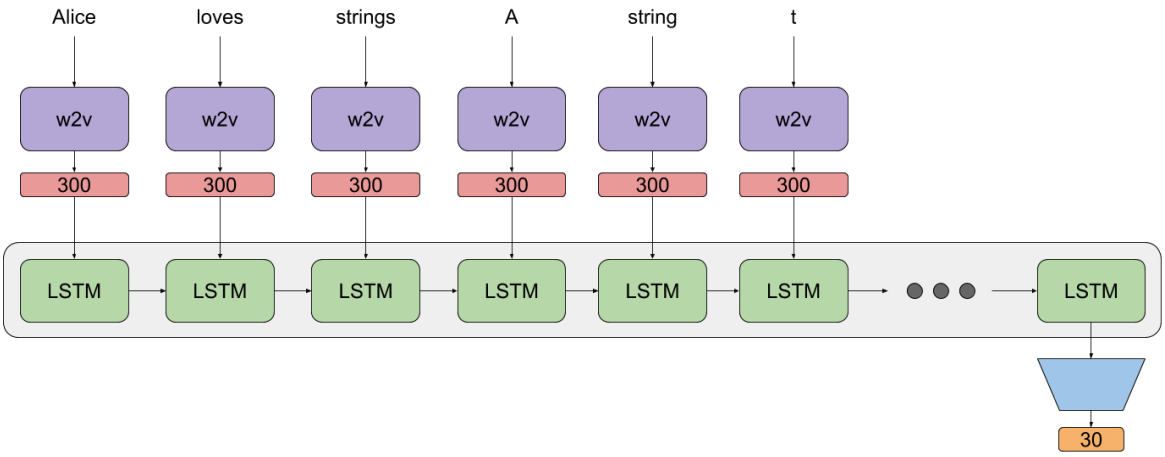

Figure 1. LSTM architecture

words part of the generated vocabulary. The words chosen in the vocabulary are those with the maximum frequency count across the collection of documents. In our project, we experimented with vocabulary sizes of 1000 and 5000. We call this representation 'hash-stream'.

5.2. Bag of Words

Once we get the hash stream of integers for each word, we can count the number of occurrences of a particular hash token. We represent these counts as a vector of dimension equal to the vocabulary size and call it 'bow' standing for bag of words.

5.3. Word-2-Vec

In this pipeline, we use the a word embedding model pre-trained on Google News dataset of about 100 billion words [1]. The model contains a mapping for about 3 million words to 300-dimensional vectors. Thus, the feature obtained for every problem is the sequence of word2vec vectors. If a particular word is out of vocabulary, i.e. if a word2vec mapping for that word does not exist, then we simply ignore those words. This representation will be referred to as 'word2vec'

'hash-stream' and 'word2vec' are richer representations than 'bow' features because they capture the temporal structure in the data. On the other hand, they are of variable length, and hence more cumbersome to handle.

[1] https://code.google.com/archive/p/word2vec/

6. Algorithmic approaches

6.1. Baselines : Approach Prediction

1. Random Forest: Random Forests trained with *entropy* criterion on 'bow' representation.
2. Gradient Boosting: Standard gradient boosting (XGBoost) on the 'bow' representation.
3. Logistic Regression: l-2 regularized logistic regression on 'bow' representation. We used lbfgs solver to speed up the convergence.

6.2. LSTM : Approach prediction

To handle variable length features as given by 'hash-stream' or by 'word2vec' , we use a LSTM model. Our architecture is shown in Fig 1. Firstly, the words are individually and independently encoded. For the hash-stream, the words are encoded as one-hot vectors over the vocabulary. For the word2vec model, we use the 300-dimensional representation directly. These are then fed to the LSTM model. We tried different number of hidden units (see Results section). There is no loss at the output of the LSTM at all but the last step. At the last time-step, we apply a linear readout layer from the LSTM states to 30-dimensional vector corresponding to the 30 output categories, followed by a softmax. This gives a probability distribution over the algorithmic categories. We use negative log-likelihood of the training data as our loss.

6.3. LSTM : Difficulty prediction

As explained in the introduction section, we also tried to solve the auxiliary task of predicting the problem difficulty

from its natural language description. The LSTM architecture is the same as the one for approach prediction with the only difference being the replacement of the readout layer (30 dimensional output linear layer followed by 30-way softmax) at the end by a smaller one with 5 dimensional output linear layer followed by 5 way softmax, corresponding to the 5 difficulty categories.

6.4. LSTM : Pre-training

Since we did not have a lot of data for the approach prediction task, we propose a novel pre-training method.

We train a LSTM with the same number of hidden units as desired finally for the approach prediction task. However, the final readout layer is the same as that for difficulty prediction. We train this network on the difficulty prediction task. Once that is done, we use the internal LSTM-LSTM weights as initialization for the approach prediction task and fine-tune from there.

The basic intuition behind this is that it is helpful to know the algorithmic category of a problem for predicting its difficulty (human consensus of difficulty). This is because problems certain categories such as *Dynamic Programming* are generally considered *difficult* problems. Hence, the hope is that in learning to predict the problem difficulty, the LSTM will also learn some information about it algorithmic category since algorithmic category is a useful cue for determining difficulty.

7. Training Details

We use negative log likelihood loss for approach as well as difficulty prediction. While training we get the gradient of this loss w.r.t. model parameters using standard back-propagation algorithm. Since the amount of data we have is quite small, we can use gradient descent instead of stochastic gradient descent. This gives much cleaner training curves. We tune the learning rate by cross validation, and we use early stopping for regularization.

8. Experiments and Results

8.1. Approach Prediction : No pre-training

Baselines : Our first baseline is the percentage of problems in the majority class. This is the best output predictable in absence of knowledge about the input. This gives 17.4% accuracy.

The following baseline algorithms run quite fast and hence we did stratified 5-fold cross validation to fit the hyper-parameters. For each algorithm we report the accuracy on a hold-out test set for the best hyper-parameters found using cross validation

1. Random Forest : accuracy = 20.92%
2. XGBoost : accuracy = 9.5%
3. Logistic Regression : accuracy = 17.34%

LSTM :

1. hash-stream, vocabulary size = 5000, LSTM states = 5000 : accuracy = 14.79%. The training curves are in Fig 2
2. word2vec, LSTM states = 300, accuracy = 15.54%.

8.2. Difficulty prediction

Baselines : Percentage of problems in the majority category is again our baseline. This gives 33.75% accuracy on the hold out test-set. The training curves are in Fig 3.

8.3. Approach Prediction : Fine-tuning

The replaced readout layer is initialized randomly. We obtain 15.54% accuracy on the hold out test set. The training curves are in Fig 4.

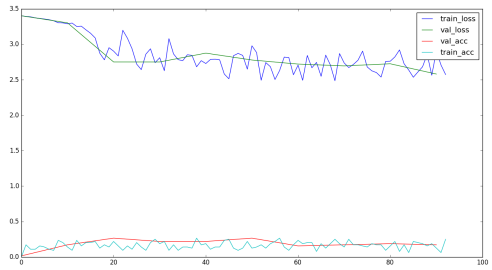

Figure 2. Approach prediction : One Hot Encoded Features

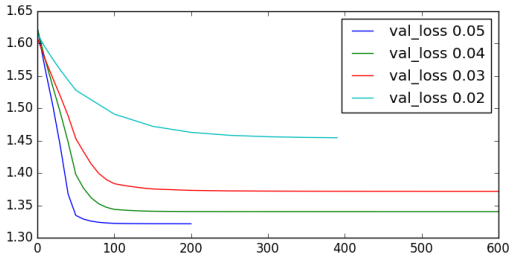

Figure 3. Difficulty prediction : validation

9. Analysis

We observe that most algorithms are doing only marginally better than the majority prediction. This is true even for dif-

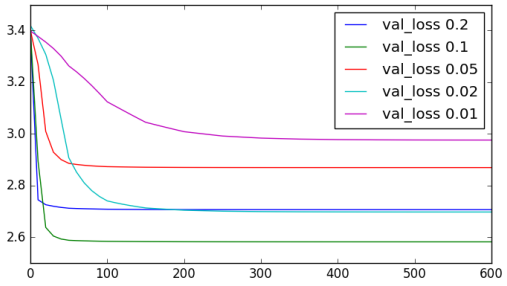

Figure 4. Fine-tuning : validation

ficulty prediction task. A major reason for this is that the problem is fundamentally hard, even for trained domain expert humans. Thus, we either need a lot of data to either to find all the regularities and predict correctly, or we need a better starting point for our model. Training a more complicated model with the current data is infeasible due to limited dataset size.

10. Future Work

We think the following ideas / approaches might help for this task.

- Most mathematical symbols and single letter variable names are ignored by the word2vec model. Either explicitly modeling them, say as one-hot encoded over a fixed vocabulary (i.e. you give one in the first position when you see a variable. If you see a new variable, give one in the second position. Now if you see the first variable ever again, it will always get the first position turned on, and so on) might help the model keep track of all the math symbols it has seen and find some relationships.
- There is data available on the accuracy of each problem on `www.codechef.com`. Many other websites also provide this information. Thus, another approach can be pre-train to predict the accuracy of human submissions on a problem which are indicative of the difficulty of the problem.
- One can also pre-train to predict the next word in the problem. This is unsupervised training since we do not need explicit labels for the problems.
- We can also use sentence2vec to get a sentence level descriptor and train LSTM on that. This will be very fast and memory efficient to train due to small number of time steps. Thus, we can also afford larger LSTM hidden units which will allow for more complex inference. The only issue is that the sentence2vec representation might not capture nuanced information which is important for our task.
- Obviously, more data never hurts. There are plenty of other websites like `www.topcoder.com`, `www.spoj.com` which host plenty of data, although very few provide actual tags.

11. Conclusion

We considered the problem of predicting algorithmic approach most useful for solving algorithmic problems given only a natural language, riddle-like problem description. This problem is highly non-obvious and stands in a stark contrast as compared to most other problems in NLP. We created two datasets for this prediction problem. We proposed and evaluated several approaches. Our experiments indicate that this is a very hard problem, and our results are just better than the majority class prediction. We expect that a deeper understanding of the text and algorithms is necessary and we propose several promising directions for future work.

References

[1] B. Amnueypornsakul and S. Bhat. Machine-guided solution to mathematical word problems. 2014.

[2] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[3] N. Kushman, Y. Artzi, L. Zettlemoyer, and R. Barzilay. Learning to automatically solve algebra word problems. Association for Computational Linguistics, 2014.

[4] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[5] H. T. Siegelmann. Computation beyond the turing limit. *Science*, 268(5210):545–548, 1995.

[6] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[7] S. M. Veres. Natural language programming of agents and robotic devices. 2008.

[8] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.