Quora Question Pairs Project Report

The Refugees 21 June 2017

Csaba Bálint, Dennis Doerrich, Paula Gombar, Anna Tőkés

1 Introduction

1.1 Problem Description

Quora Question Pairs is a challenge available on Kaggle. The goal of the challenge is to detect whether question pairs are duplicate or not. Currently, Quora uses a Random Forest model to identify duplicate questions. The ground truth is labeled by human experts and is either 0 or 1.

For each ID in the test set, we needed to predict the probability that the questions are duplicates. Submissions were evaluated on the log loss between the predicted values and the ground truth.

1.2 Data Analysis

The data is provided in CSV format and contains the following fields:

1. id - the id of a training set question pair
2. qid1, qid2 - unique ids of each question (only available in train.csv)
3. question1, question2 - the full text of each question
4. is_duplicate - the target variable, set to 1 if question1 and question2 have essentially the same meaning, and 0 otherwise.

We were given two data sets, train and test. The train test contains 404290 question pairs, of which 36.92% are duplicate. The test set contains 2345796 question pairs, almost 6 times more than the train set. This is because Kaggle has supplemented the test set with computer-generated question pairs as an anti-cheating measure.

2 Approaches

2.1 Data processing

We tokenize the text of a question by splitting it into individual words and representing them in lowercase. We perform stop-word removal with regards to the stop-word dictionary found in the Natural Language Toolkit for English. However, we do not remove possibly indicative words such as "what", "how", "where", "why", and "who".

2.2 Features

Padded Sequences We convert every question to an integer array, where the integer represents the index of the word in the dictionary. We 'pad' these sequences in order to have the same length, as we later want to feed the sequences to a classifier or a neural network. Padding amounts to cutting off at a certain fixed length if the question is too long, or adding zeros if it is too short.

Word Embeddings Instead of using a pre-trained model, we trained Google's skip-gram model on our datasets, both train and test. We used vectors of the size 300, context size 10, the skip-gram model, hierarchical softmax as the training algorithm, and used 40 for the minimum word count. We define question embedding as the mean vector of the embeddings of its words.

Embedding Matrix The word embedding matrix is generated by assigning each integer value from the padded sequences its respective vector. This is done to combine generated word embeddings and the weights in the neural network. The embedding matrix serves as an embedding layer for the questions.

WordNet We constructed features using WordNet, a large lexical database of English. Here, nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. We used the Leacock-Chodorow similarity to calculate question pair similarity. It returns a score denoting how similar two word senses are, based on the shortest path that connects the senses and the maximum depth of the taxonomy in which the senses occur. The relationship is given as $-\log(p/2d)$ where p is the shortest path length and d the taxonomy depth. We created a matrix of similarities between every two words occurring in the train set. A similarity score of a question pair is defined as the sum of maximal similarities between each word pair, divided by the length of the first question and the maximal value of the Leacock-Chodorow similarity.

Simple Features We constructed features capturing syntactic information from the question pairs. These features include the number of overlapping unigrams and bigrams (normal and TF-IDF-weighted), the ratio of stopwords, the cosine distance between questions, difference

in character and word count, and the number of capitalized letters.

3 Models

3.1 Word-counting approaches

As a baseline, we built two simple word-counting models. The first one uses a threshold optimized on the training set. We count how many words in common two questions have, and if the number of words in common is less than the threshold, the questions are not considered duplicates, and vice-versa. Finally, we use the best-scoring threshold to produce predictions.

In the second model, we use the number of overlapping words and average it by dividing with the question length (minimal of two options) as features for logistic regression.

3.2 Word embeddings + LSTM

Our initial neural network approach consisted of building a 3-layer deep NN that takes padded sequences as input for the embedding layer, merges them, and outputs them in the dense layer. This already yielded promising results. By adding a LSTM layer before merging and using word embeddings as weights for that layer in the form of an embedding matrix, we maintained the information of word order and linguistic context gained by word embeddings. Characteristically, after about 6 epochs an increase in validation log-loss was observed. By adding dropout and batch normalization in LSTM and dense layers we noticed that the overfitting was confined until the 10th epoch.

3.3 Simple Features + XGBoost

We used XGBoost,[1] a library for boosting trees algorithms. The underlying principle, gradient boosting, produces a prediction model in the form of an ensemble of decision trees. We used the simple syntactic features, and a 5-folded grid search to find optimal hyper-parameters of the model.

4 Predictions

As the final prediction model, we used the best-performing classifier, which is XGBoost trained with simple features. In addition to this, we also feed the embedding matrix learned in the LSTM neural network as features. We perform under-sampling of the negative class and use 5-fold grid search to tune the hyper-parameters of the model. Scores of this model, and other relevant models, can be seen in Table 2.

[1] https://xgboost.readthedocs.io/en/latest/

4.1 Mapping

In our attempts to achieve a better score, we undersampled the negative class and decided to apply a function to skew the final prediction values. As seen on the Kaggle discussion boards, the simplest function is multiplication by 0.75, which did yield the best result of our classifier.

However, we decided to also use another function that we can control more precisely. The function $f \in [0,1] \rightarrow [0,1]$ needed the following requirements:

- $f(0) = 0$ and $f(1) = 1$
- $f'(0) = \alpha$ the angle at $(0,0)$
- $f'(1) = \beta$ the angle at $(1,1)$

Assuming that f is a third-order polynomial in the form of $f(x) = ax^3 + bx^2 + cx + d$, one can write and readily show that the solution is the following:

$$a = \alpha + \beta - 2, \quad b = 3 - 2\alpha - \beta, \quad c = \alpha, \quad d = 0.$$

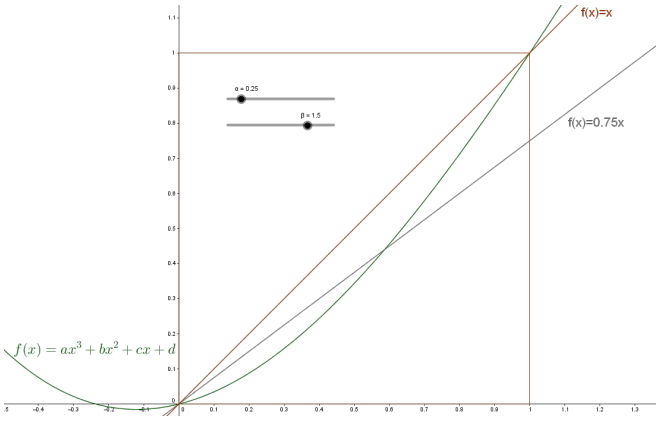

Figure 1: Mappings used on predictions to increase Kaggle score.

This allows us to imagine a $[0,1] \rightarrow [0,1]$ mapping, and implement it using only two variables. For example, when $\alpha = \beta = 1$, the function is the identity mapping $f(x) = x$. When $\alpha = \beta = 0$, we have a standard smoothing function often used for increasing contrast.

We are aware that this can be done more elegantly using Lagrange interpolation, but due to the task's simplicity this was the easier approach. We also added a multiplier hyper-parameter, denoted κ later, because by carefully setting the $\alpha = 0.25$, and $\beta = 1.5$, the Kaggle score did not improve.

4.2 Kaggle-score Optimization

We had 6 hyper-parameters that we could easily change, and the result on Kaggle depended on them dramatically. A decision was made to optimize it, while keeping in mind that we only had five submission to do so.

Step	Log-loss	α	β	ETA	κ	p	ROUNDS
1	0.58300	0.2500	1.5000	0.0500	0.8000	0.1650	250.0000
2	0.68396	1.0000	1.0000	0.0700	1.0000	0.2500	1500.0000
3	0.58200	1.8000	0.2000	0.1000	0.6000	0.4000	100.0000
4	0.63620	0.3099	2.9774	0.0779	1.0327	0.1266	152.0000

Table 1: Scores with parameters using Bayesian optimization for better Kaggle score. The first two lines are based our guesses for best parameters, and the last two were made by the Gaussian process-based optimization algorithm. Unfortunately, our concatenated features produced worse results, so the optimization did not correct it. The p is a parameter for negative oversampling, and α, β, γ are for the mapping of probabilities before submission to decrease log-loss.

Therefore, we decided to use a Bayesian optimization algorithm [5], that uses Gaussian processes to model the function to be optimized, while it takes very few samples. The results of this approach are shown in Table 1.

Model	Features	Score
All Negative	-	6.01888
Word-count thres.	Word overlap	10.54367
Word-count log-reg	Word overlap	0.49937
Simple NN	Padded	0.44419
LSTM	Word embeddings	0.41466
XGBoost *0.75	Simple	**0.34976**
XGBoost	Simple	0.35195
XGBoost *0.75	WordNet + Simple	0.35494
XGBoost + All	All	0.58200

Table 2: Kaggle public scores, expressed as log-loss. "thres" stands for threshold and "log-reg" stands for logistic regression. Surprisingly, the best score was achieved using the most simple features.

5 Discussion

Our best-performing model is XGBoost trained for 315 rounds, the learning parameter is set to 0.11, the features used are just the simple syntactic ones, and the outputs are multiplied with 0.75. This produces a public score of 0.35, which is much better than the baselines of simple word-counting models, 10.54 (threshold optimization) and 0.5 (logistic regression).

Quite surprisingly, we found that introducing the weights learned in the embedding layer of LSTM as additional features did not help at all. On the contrary, the scores produced with this model ranged from 0.58 to 0.68.

Also not expected was the sub-par performance of our neural network approach. The combination of word embeddings and LSTM using dropout scored 0.415.

In general, models in which we under-sampled the negative class performed better, and hyper-parameter optimization did not significantly improve the model's scores. Finally, it seems that simple syntactic features are more powerful in capturing question duplicity than more sophisticated ones.

6 Contributions

Our team consists of four team members. The contributions of individual team members are listed below, sorted alphabetically.

Csaba Bálint Contributions:

- Created the preprocessing files
- Worked on word embeddings for a short time
- Worked on RNN features that were abandoned due to time limits
- Mapped probabilities with a cubic function
- Bayesian Optimization on scores using GP-s

Dennis Doerrich Word embeddings + LSTM and simple neural network approach using the embedding matrix and padded sequences.

Paula Gombar Produced word embeddings on both datasets with word2vec tool and generated question embeddings, implemented word-counting models, implemented simple features and XGBoost pipeline (feature generation, model training and cross-validation with 5-fold grid search).

Anna Tőkés Implemented the WordNet feature:

- created the dictionaries from the words appear in the train set;
- created the similarity matrices based on the lch_similarity metrics;
- calculated the similarities for each question pair.

Source Code Implementation is in Python 2.7 using scikit-learn and Tensorflow. The source code can be found on GitHub[2].

[2] https://github.com/DonatDonat123/QUORA

References

[1] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. "Learning Word Vectors for Sentiment Analysis." The 49th Annual Meeting of the Association for Computational Linguistics. ACL 2011.

[2] Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems. 2013.

[3] Maas, Andrew L., et al. "Learning word vectors for sentiment analysis." Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1. ACL 2011.

[4] Samuel Fernando and Mark Stevenson. "A Semantic Similarity Approach to Paraphrase Detection". 11th Annual Research Colloquium of the UK Special-interest group for Computational Linguistics. 2008.

[5] Eric Brochu, Vlad M. Cora, and Nando de Freitas., "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning"., CoRR, abs/1012.2599,2010,