Quora Question Pairs

BHAVIKA TEKWANI, George Mason University

Quora is a quasi-forum designed to crowdsource knowledge and opinions on a wide range of topics. In the year 2016, there there were 29,000 questions[1] asked about the US Presidential Election alone. Other popular topics included outer space, artificial intelligence, Brexit and the Rio Olympics. Needless to say, the amount of content on Quora is extremely vast. Often, users tend to ask a question that has already been asked and answered. On Quora, this means users spend more time waiting for answers that must have already been answered but are difficult to discover through search functions. Quora has enabled several users to become influencers on topics of interest to them based on how many questions they answer in their area of expertise. For these users, identifying identical questions may save them time in answering and improve the quality of answers. The problem we try to solve in the Quora Question Pairs Kaggle competition is that of duplicate question identification. We are interested in identifying questions with the same intent and quantifying this with a value between 0 and 1.

CCS Concepts: • **Natural Language Processing** → Semantic Analysis; **Data Mining**

KEYWORDS

text classification, data mining

1 INTRODUCTION

Quora Question Pairs is a Kaggle competition where we are provided with pairs of questions asked by users on Quora. We are tasked with predicting the probability that any two questions have the same intent and are duplicates. Often, two questions may sound similar in terms of the general topic they belong to, however, the answers may be vastly different. Take for example, the questions “Are dogs allergic to chocolate?” and “Why are dogs allergic to chocolate?”. For humans, it is easy to identify that one of these can be answered with a yes or no and the other requires an elaborate response. The problem lies in understanding natural language rather than merely processing it. We will explain some strategies for predicting the extent to which two questions may be similar in the following sections.

First, let us take a look at the data. We are given two datasets - a train and test set. The train dataset contains the following fields - id, qid1, qid2, question1, question2 and is_duplicate. qid1 and qid2 represent the unique ID for questions on Quora whereas id is merely an identifier for a pair. is_duplicate contains a value that may be 0 (not duplicates) or 1 (duplicate).

The test set is similar, but it contains only test_id, question1 and question2. We have to predict the value of is_duplicate for each pair of questions in the test set. This value must be in between 0 and 1.

Train	Test
id	test_id
qid1	question1 [text]
qid2	question2 [text]
question1 [text]	
question2 [text]	
is_duplicate [0, 1]	

Table 1. Train and test data snapshot

The train dataset has 404289 cases and the test set has 2345795 cases. Thus, the test set is approximately 5 times larger than the train set.

We observe the distribution of duplicate question pairs in the train dataset. Only 37% of the train examples are marked as duplicates. Thus, we know that we have an imbalanced data problem.

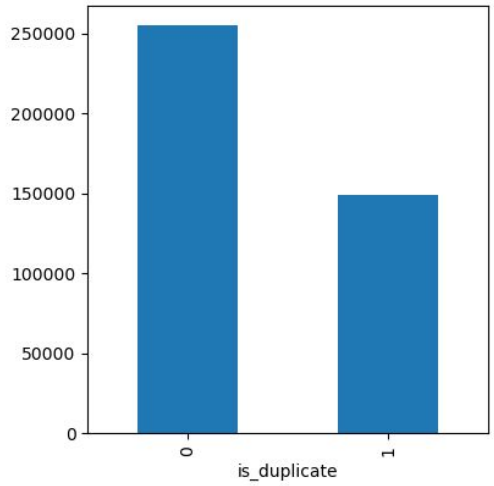

Fig. 1 The distribution of is_duplicate labels (1 indicates duplicate) in the train set

The Kaggle leaderboard evaluates submissions only on 35% of the dataset. The metric used in this competition is log loss (Equation 1). The goal is to minimize the log loss as we make predictions for is_duplicate.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(p_{i,j})$$

Equation 1. Log loss metric

In order to avoid hand-labeling of the dataset and gaming of the leaderboard, Quora has introduced some computer generated question pairs in the test dataset. We now look at ideas for feature generation and modeling.

The text in both train and test sets has been preprocessed before feature extraction.

The steps involved in preprocessing are:

1. Removing inconsistencies in data: We replace instances of “9 11” with “911”, “dms” with “direct messages”, “USA” with “America”. This is mostly done to remove common mistakes that users make either in grammar or in spelling. The list of corrections has been adopted from user ‘currie32’ on Kaggle. I’ve added some common spelling errors to this list and processed the text accordingly.
2. Stopword removal: We use NLTK’s stopword list and add some stopwords of our own. We filter out any occurrence of these words from the question pairs.
3. Remove punctuation.
4. Stemming: We use the Porter Stemmer in NLTK to replace each word with its root word. This reduces the size of our corpus and makes the process of identifying questions that mean the same thing but vary in tense easier.

When missing values were found in the question pairs, we replaced them with empty strings but they are not excluded from analysis (i.e., they are still in the training set).

2 EXPERIMENTAL AND COMPUTATIONAL DETAILS

2.1 Computation

We ran all our experiments on a single c4.8xlarge AWS instance. This instance had 36 vCPUs and 60 GB memory. Additionally, we stored all our data and computations to disk on a 100 GB EBS Volume. Using an AWS instance directly enabled us to perform the experiments detailed later on.

2.2 Feature Engineering

Based on the nature of the text data we have, which is short questions and not long paragraphs on inter-related documents, we find that some very simple summary statistics and metrics may be helpful as features. We calculate some features that will be used in the models and some which while created for analysis or use in deep neural networks, didn't quite fit well into the structure required by these neural networks. Nevertheless, we will discuss all types of features built for analysis.

Let us label each featureset combining related concepts as FS-1, FS-2, FS-3, etc. Broadly, we have the following types of features - lengths of questions, word counts and character counts; fuzzy text features; wordshare index; word mover's distance and normalized word mover's distance; Sent2Vec features and a set of distance metrics, skew and kurtosis between the Sent2Vec representation of question pairs. We summarize the features in Table 2.

Featureset	Features
FS-1	Length of question 1 & 2 - len_q1, len_q2
	Length of question 1 & 2 after cleaning - len_q1c, len_q2c
	No. of words in question 1 & 2 after cleaning - words_q1c, words_q2c
	No. of characters in question 1 & 2 after cleaning - chars_q1c, chars_q2c
	Wordshare index (no of common words between question 1 & 2, normalized)
FS-2	Fuzzy features - QRatio, WRatio, Partial Ratio, Partial Tokenset Ratio, Tokenset Ratio, Partial Tokensort Ratio
FS-3	Word Mover's Distance & Normalized Word Mover's Distance
FS-4	Distance metrics - Cosine, Cityblock, Canberra, Euclidean, Minkowski, Braycurtis, Jaccard, skew of question 1 & 2 vectors.

Table 2. Featuresets created for analysis

Sent2Vec: In a Word2Vec model, each word is converted to a single dimensional vector with 300 elements. We use a similar logic and extend it to sentences as a whole. In our dataset, we consider each question in a question pair as a sentence and generate a vector representation (embedding) for it. The algorithm is as follows and was first introduced by Pagliardini et al.[3] as an unsupervised model.

While we write our own implementation of Sent2Vec, we are using Google's pretrained Word2Vec model to extract word embeddings.

Pseudocode for Sent2Vec

```
function sent2vec:
      words = word_tokenize(s)
      words = remove_stopwords(words)

      M = [ ]
      for each word in words:
            word_vec = get_wordvector(word)
            M.append(word_vec)

      for each word_vec in M:
            // Add up 300 elements for each word by index
            // Add element of word 1 to element 1 of word 2 and so on...
            v += word_vec

            // Calculate the dot product of vector 'v' with itself
            sum2 = (v * v)

            // Add up all elements of sum2 and take the square root
            s = sum(sum2)

            sum_root = sqrt(s)

            return (v/sum_root)
```

The value returned from the Sent2Vec function is the sentence embedding - a 1-dimensional vector with 300 elements. We use this sentence embedding for distance calculations in FS-4. We must note that any other model like GloVe can also be used similarly for extracting word embeddings. I have not compared the performance of distance metrics when obtained via GloVe and Word2Vec, but this can be a trivial addition to the analysis too.

Word Mover's Distance (WMD): WMD is a distance function that evolved from the concept of Earth Mover's Distance. It was introduced by Kusner et al[2]. The Word Mover's Distance calculates the distance between two documents from their word embeddings. It does this by calculating the minimum distance embedded words from one document would have to travel to reach the embedded words in another document. We use the raw WMD and a normalized WMD as features for our analysis. We are using the WMD function in the *gensim* module for Python.

2.2 Feature Selection

We've constructed a total of 26 features across FS-1 to FS-4. Not all features may be equally useful, in fact it's quite likely that the more features we use, our models would overfit. We tackle this in 2 steps: first we plot the feature importance using a simple XGBoost model. XGBoost gives an F-score which is simply an indicator of how often a branch was split on each particular feature. Secondly, we use certain features to make submissions to the Kaggle leaderboard.

It is a natural but naive assumption that questions which are duplicates of each other would have a lot of words in common. We test this hypothesis by plotting the wordshare index and is_duplicate values for the test set. Fig. 2 shows that the distribution of wordshare index is higher for cases where the question pairs are duplicates.

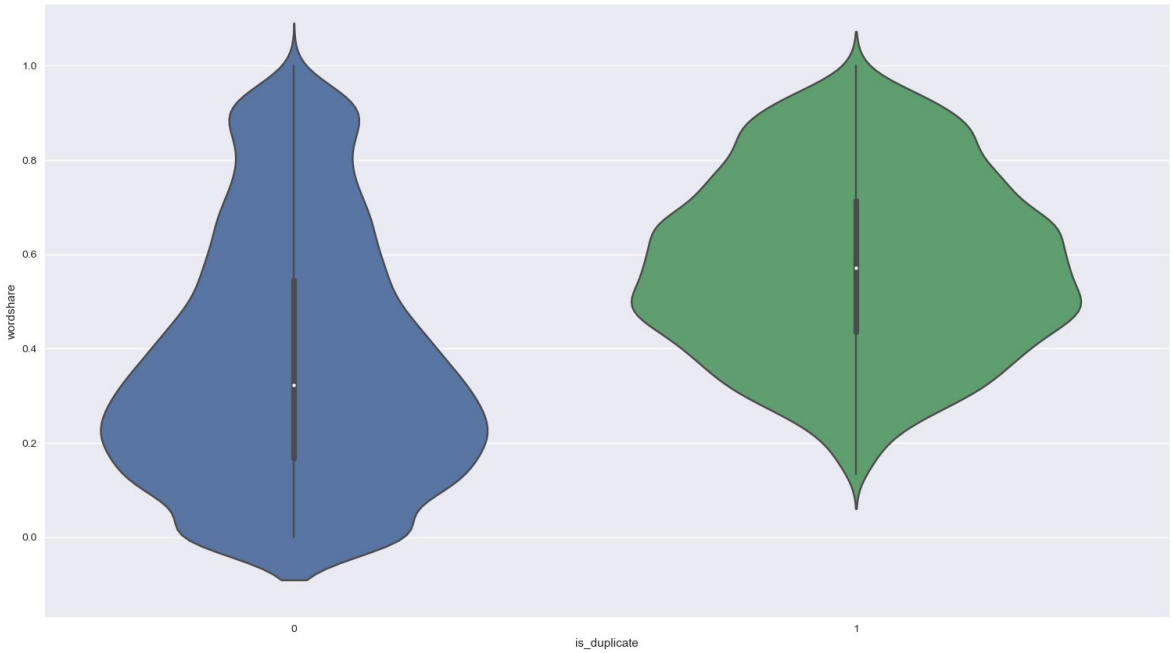

Fig. 2 Violin plot showing the distribution of wordshare index for duplicate (green) & non-duplicate (blue) question pairs

We also fit an XGBoost model on various subsets of features to gauge the most important ones based on the F-score. Fig. 3 shows the feature importance for the baseline model which used FS-1 and FS-2.

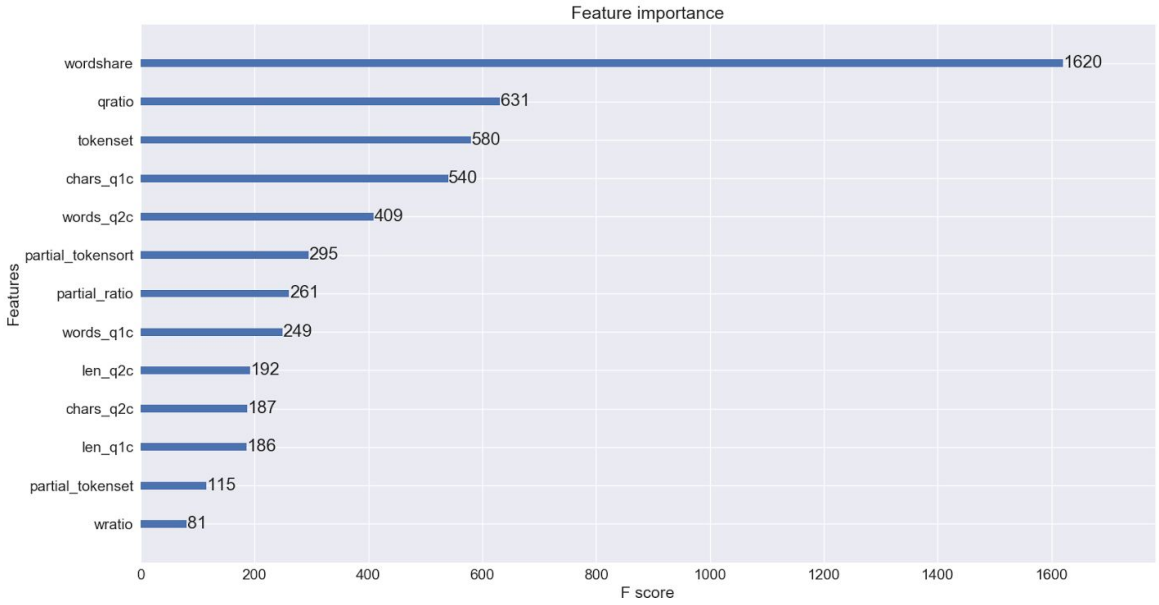

Fig. 3 Feature Importance for Baseline XGBoost model (FS-1 and FS-3, not hyperparameter optimized)

As we observe from Fig 2 and 3, wordshare is a strong indicator of question similarity. We discuss the results of using wordshare as a feature in the later section.

Next, we include Word Mover's Distance and its normalized version as features. Fig. 4 shows the relative feature importance when the same XGBoost model is fit using different features on the training set. As discussed earlier, Word Mover's Distance is successful as a distance measure since it considers document similarity in the word embeddings context. TF-IDF and Latent Semantic Indexing (LSI) are thought to be good approaches for reducing documents to vectors and calculating similarity metrics but they tend to focus more on *how often* words occur rather than *which* words they co-occur with. This seems to be an apparent advantage of pre-trained models like Word2Vec and GloVe.

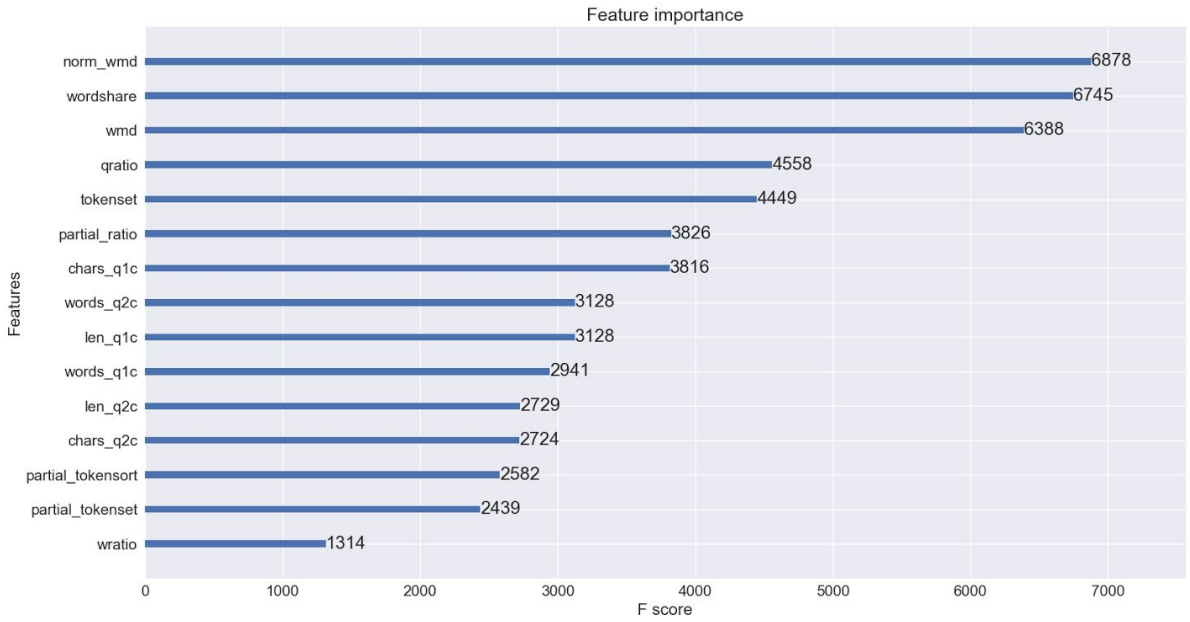

Fig. 4 Feature importance when Word Mover's Distance is included.

We also look at feature importance when all featuresets (26 features in all) are used with the same XGBoost model. Fig. 4 shows that skew and kurtosis come out on top, followed closely by wordshare and Word Mover's Distance. We discuss the disadvantages of a model using all these features in the next section.

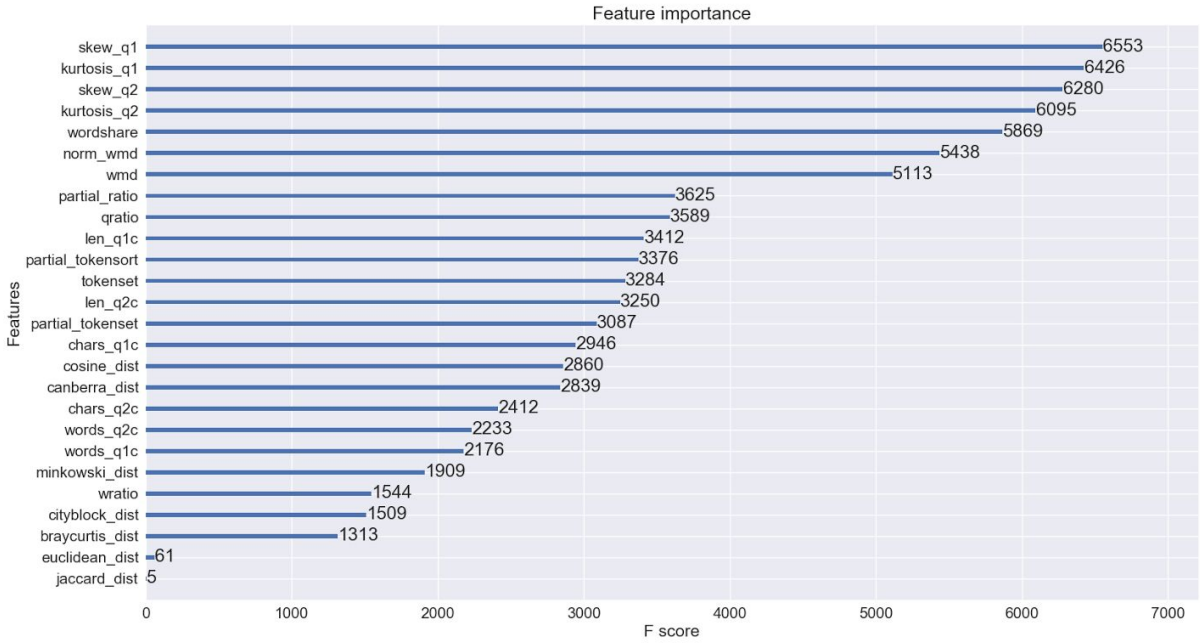

Fig. 5 Feature importance when all 4 feature sets are used

3 RESULTS AND DISCUSSION

At first, we decided to use a simple XGBoost model as a baseline for our submissions to the Kaggle leaderboard. This baseline model is not hyperparameter optimized. We use FS-1 and FS-2 as features. On submitting the predictions from this model (let's call it the XGB_Baseline), we achieve a log loss of 0.63380 on the leaderboard.

We now compare the results of all our models to only two results - the XGB_Baseline and the highest scoring log loss on the Kaggle leaderboard. On Kaggle, as of 05/09/2017, the lowest log loss is 0.11967 by the team called 'DL Guys'.

We present a summary of our models, feature combinations, log loss and rank on the leaderboard with different methods used.

Note: HP refers to Hyperparameter Optimized models. LB stands for leaderboard and refers to the score and rank on the test set as calculated on Kaggle. CV refers to cross validation - we used Stratified KFold cross validation with 10 folds for the blended model, a train-test distribution of 80-20 for XGBoost and 90-10 for LSTM. We did not use a grid search to optimize the blended model.

Model	Features	Log loss (CV)	Log loss (LB)	LB Rank
XGB_Baseline	FS-1 + FS-2	0.68795	0.63380	~1500s
XGBoost [HP optimized]	Wordshare index	0.452776	0.46868	~1300s
XGBoost [HP optimized]	Wordshare + FS-3	0.494051	0.41594	1348
Blended model - 6 regressors including 2 Random Forest regressors, 2 Extra Trees regressors, 1 Gradient Boosting Regressor and Logistic Regression	FS-1 + FS-2 + FS-3 + Wordshare		0.41859	1383
Blended model - 6 regressors including 2 Random Forest regressors, 2 Extra Trees regressors, 1 Gradient Boosting Regressor and Logistic Regression	FS-3 + Wordshare + QRatio		0.45639	1383 (Score worsened, no change in rank)
XGBoost [HP optimized]	Wordshare + FS-3 +FS-4	0.42231		
LSTM	Word2Vec embeddings	0.2891	0.31858	**638***
XGBoost [HP optimized]	FS-1 + FS-2 + FS-3 + FS-4	0.40698	0.42231	638 (Score worsened, no change in rank)

Table 3 : Experiment results with 3 types of models - XGBoost, a blended model and a DeepNet (LSTM)

As can be seen from the results above, the LSTM model gives us the biggest boost in performance. The architecture for this deep network is described below. Similar models have been built in the Keras documentation and the architecture I use is not something I came up with, rather I've borrowed from examples in the Keras documentation[4,5], models used in other text related tasks on Kaggle and this sequence classification tutorial[6].

The LSTM has a primary Embedding layer which is where we pass it an embedding matrix with a maximum no of words set to 200,000. The embedding dimensions are set to 300 (recall that Word2Vec creates 1d vectors of 300 elements for each word). We restrict the sequence length to 30 but this is arbitrarily chosen and can be varied. The sequences are built using Keras' Tokenizer. There are 2 dropout layers, 2 BatchNormalization layers and 1 Dense layer at the end where a sigmoid activation function is used. We did not try other parameters for the activation function. We're using callbacks offered by Keras for early-stopping. The validation set is 0.1 of the original dataset and we are shuffling the dataset for each epoch. While epochs are set to 200, we observe that the model converges between 23-27 epochs on multiple runs. To deal with the class imbalance, we assign weights to the classes - again, these values have not been varied since the training time for each run of the LSTM is over 10 hours.

For the XGBoost model, the hyperparameters we obtained by GridSearch are as follows:

```
n_estimators: 500,
subsample: 0.9
learning_rate: 0.05
max_depth: 9
colsample_bylevel: 1.0
```

Changing the n_estimators and max_depth made the most difference in the log loss. We saw that Fig 5 ranked skew and kurtosis features very highly in terms of importance, however, the corresponding model - XGBoost with four featuresets does not perform very well (0.42231 on the test set). It may be reasonable to assume that this model is overfitting to the train data. Using all features makes the model complex and unable to generalize.

The blended model consisting of 2 Random Forest regressors, 2 Extra Trees regressors, Logistic Regression, Gradient Boosting regressor was not hyperparameter optimized. We used the default hyperparameters but with the criterion as "mean squared error" for each.

4 CONCLUSIONS

While the model we built is indeed not very complex and nuanced in terms of natural language understanding, it provides decent results. We observe that Word2Vec performs very well with a pre-trained model too and with more compute power, we would like to add more words to the corpus and see the results. We've used XGBoost and other tree based regressors mainly because they're very fast to train and XGBoost is a very popular choice for Kaggle competitions. This was a good opportunity to try out a simple deep learning model and we would conclude that coming up with good deep learning architectures is difficult without a deep understanding of the drawbacks of certain methods especially because of the large training time and the vanishing & exploding gradient problem. We wanted to try out GRUs and build features based on part-of-speech tags but I realized incorporating those into the existing architecture was not straightforward and at times, not recommended. Overall, we think #638 is a decent rank for my first attempt at a Kaggle competition and LSTMs but we would have liked to use Sent2Vec and POS tag features in the LSTM somehow.

ACKNOWLEDGMENTS

User ‘currie32’ on Kaggle for their list of word replacements towards the goal of cleaning up text.

REFERENCES

[1] A Year in Questions - Quora blog, https://qph.ec.quoracdn.net/main-qimg-5799e66af99cd2e0d6f22f3875cc92e4.webp Last accessed: 05/08/2017

[2] Matt Kusner et al., “From Word Embeddings to Document Distances,” in *International Conference on Machine Learning*, 2015, 957–966, http://www.jmlr.org/proceedings/papers/v37/kusnerb15.pdf.

[3] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi, “Unsupervised Learning of Sentence Embeddings Using Compositional N-Gram Features,” *arXiv:1703.02507 [Cs]*, March 7, 2017, http://arxiv.org/abs/1703.02507.

[4] Guide to the Sequential Model - Keras Documentation. Accessed May 9, 2017. https://keras.io/getting-started/sequential-model-guide/.

[5] “Fchollet/keras.” *GitHub*. Accessed May 9, 2017. https://github.com/fchollet/keras.

[6] Brownlee, Jason. “Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras.” Machine Learning Mastery, http://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/.