Sarmad Tanwir

**Identifying Question Pairs with the Same Intent in the Quora Questions Pair Corpus**

**Introduction**

Quora is a website where people can post their questions and then those questions are answered by the others visiting Quora. It is a place to seek and share knowledge where you can ask anything about anything and invariably will find someone who will answer your question. Since the website is that flexible, it happens frequently that the same questions are asked multiple number of times. Over 100 million people visit the website every month.

Multiple questions with the same intent are a problem for both the seekers and writers. It is because people who are looking for answers will have to scan through all similar questions to find the best answer. Also the people who write answers to the questions will prefer to write their answers only once. It is hard for anyone to keep answering the same questions so long as people keep asking them again and again.

In this project, we’re looking to explore different ways that this problem can be solved and see which features and models give us the best results.

**Dataset**

The dataset comes from Kaggle (supplied by Quora) [1] and is publically downloadable. There are two files test.csv.zip and train.csv.zip, which contain the test and train data sets respectively. The training dataset contains actual questions from Quora that are provided in pairs and are labelled as duplicate or not duplicate by human readers. The test set, on the other hand, contains both actual questions and a lot many computer-generated question pairs. The intent of including the computer-generated question pairs is to make cheating harder in the competition. The computer generated question pairs are not counted in the scoring. This is to avoid the overfitting of test data by the competitors.

The competition is poses a classification problem. Both the test and training sets have question pairs labelled by human readers. Our job is to train a classifier over features generated from the training set that can accurately predict the labels in the test set.

**Data fields:**

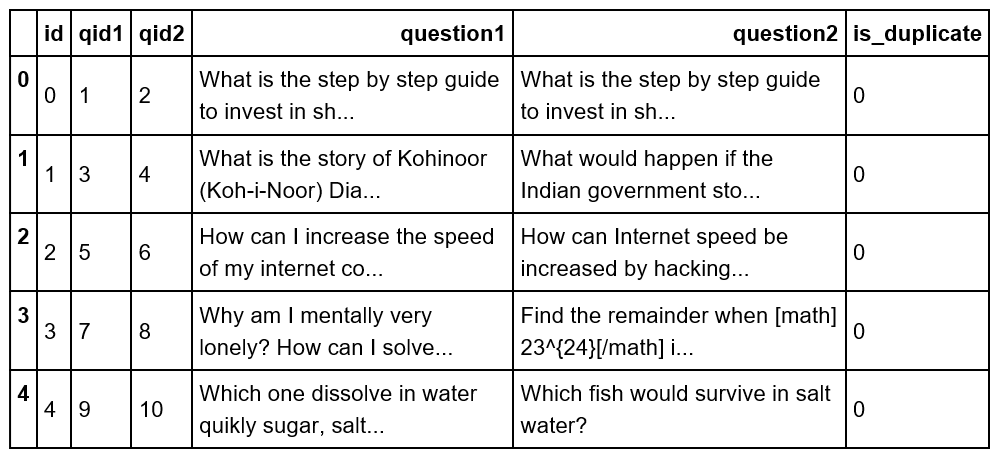
The following data fields are provided in the training and test set.

**Table 1: List of Columns in Provided Dataset**

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Description** |
| 1 | Id | The id of a training set question pair |
| 2 | Qid1 | Unique id for question 1 (only available in training set) |
| 3 | Qid2 | Unique id for question 2 (-do-) |
| 4 | Question1 | The text of question 1 |
| 5 | Question2 | The text of question 2 |
| 6 | Is\_duplicate | The target variable |

**Exploring the Dataset**

We take an exploratory look at the dataset. The top 5 rows of the dataset look as follows: -



The total number of rows in the dataset is 404290. Distribution by the class variable “is\_duplicate” is shown by the following barchart.

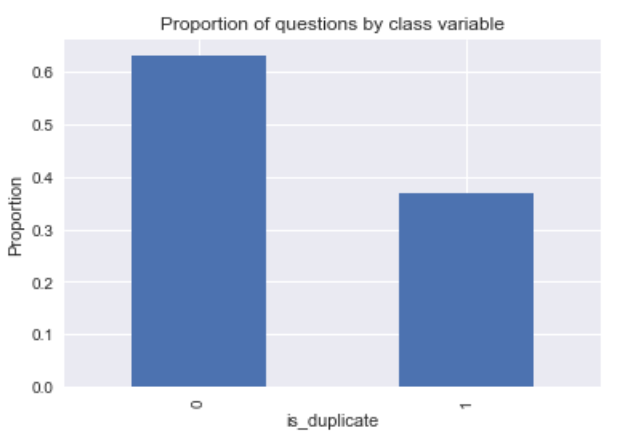


Fig 1: Proportion of duplicate and non-duplicate pairs in the training set

After having an initial look at the dataset, we go ahead with the preprocessing using state-of-the-art NLP libraries spaCy [2] and genism [3]. We’ll further explore the dataset after we have NLP and statistical features.

**Preprocessing**

We preprocess the text of the questions in the following steps: -

1. UTF Encoding

We covert all text to utf-8 for ease in future processing.

1. Lemmatize and Remove Punctuation

We lemmatize using nlp parsing in spaCy. Then, we remove the punctuation and spaces. Finally, we join all lemmatized words in each question by inserting spaces.

A sample of lemmatized questions is provided below.

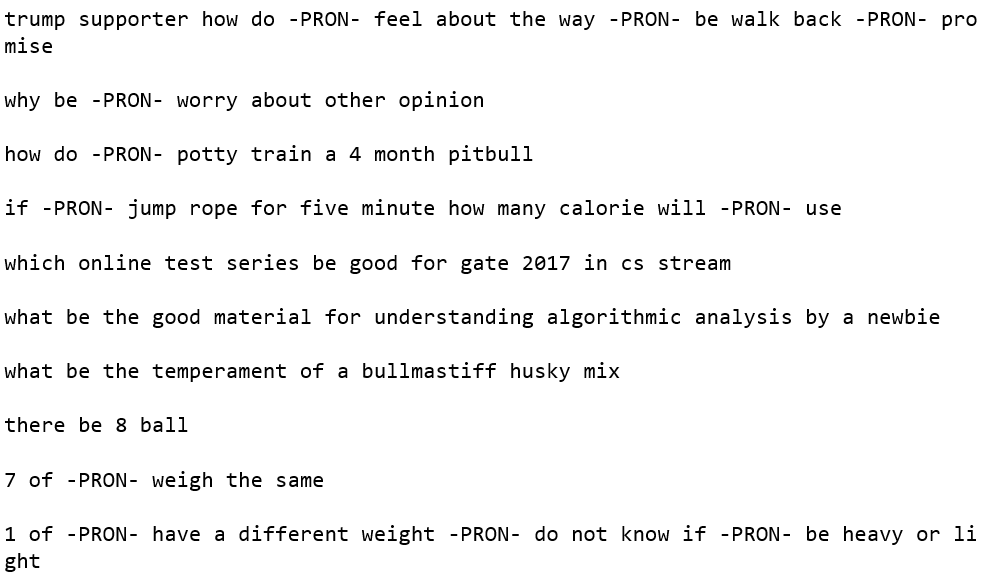


Fig 2: Lemmatized Questions

We observe that all pronouns have been replaced by the token –PRON-. This is a new feature in spaCy. Also all verbs have been converted into their basic form e.g. ‘is/are’ are changed to ‘be’.

1. Learn bigrams and trigrams in the text

We process the entire corpus using the Phrases function from genism to learn the bigrams in the first pass and save the model. We use the bigram model to transform each question and save the questions with bigrams joined with a hyphen. Then, we learn the trigrams and use the model to transform each question. We save the entire corpus.

The trigram-ed representation of the questions in Fig 2 is show in Fig 3 below: -

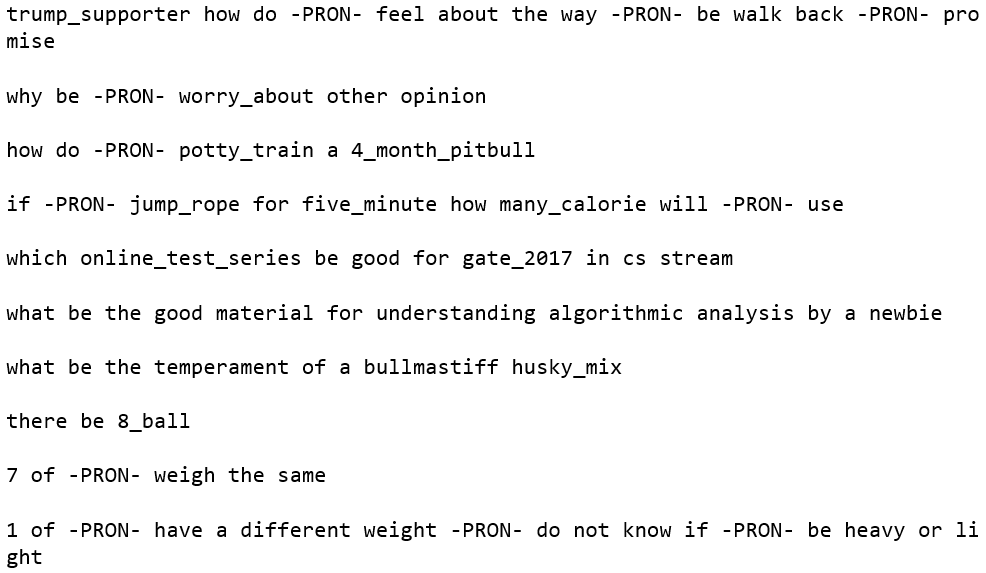


Fig 3: Lemmatized questions with trigrams identified

In Fig 3, we can see bigrams such as trump\_supporter and potty\_train and a trigram online\_test\_series.

1. Remove stop words

After learning and identifying the trigrams, we remove the stop words from the corpus by filtering using the spaCy’s English stop words list and then save the entire corpus for further processing. After removing the stop words, the above questions now look like this.

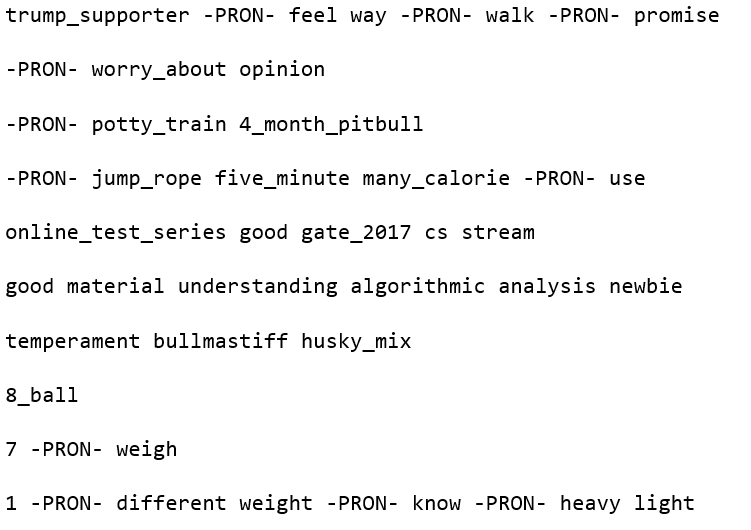


Fig 4: Trigram corpus with stop words removed

With this corpus, we proceed to generate features that will help us build classification models.

**Model Building**

**Feature Generation**

**Vector Representations**

We use the saved tri-gram corpus to generate a bag of words representation and save the representation to the disk. We also build a Tfidf [4] representation from the bag of words representation. We use gensim’s doc2bow() and TfidfModel() functions to generate these representations.

**LDA/LSI**

We employ topic learning using Latent Dirichlet Allocation (LDA) [5] and Latent Sematic Indexing (LSI) [6]. For our exploratory analysis, we learn 50 LDA topics from the entire questions pairs corpus. For the final analysis, we learn 300topics. To generate features, we apply the learned models to each question and save the resulting vector embeddings in a csv file.

For the LDA, these vector representations consist of the probabilities of each document belonging to a particular topic. An example of this representation for a random question is provided below.

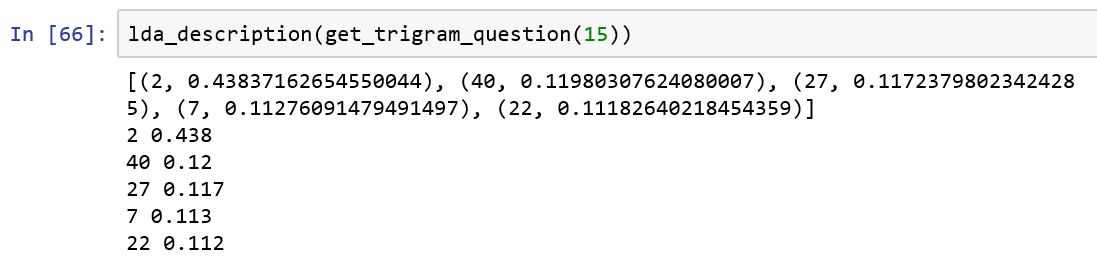


Fig 5: LDA description of a question

In the above example 2, 40, 27, … etc. are the topic numbers. The probability of the question belonging to that topic is the provided as the second element in the tuples.

The LSI provides a lower dimensional representation of each question derived from a low-rank approximation of the term-document (word-question) matrix. The elements of the LSI vectors cannot be interpreted as probability as in the case of LDA. More details can be found at [6].

Dictionary Generation

For learning both LSI and LDA models, we build a dictionary of all the tokens (words) in the corpus including the bigrams and trigrams. After building this dictionary, we filter tokens that are very rare or too common using the filter\_extremes() function (with no\_below=10 and no\_above=0.4) to make sure that they do not adversely affect the topic learning process. After removing these tokens we reassign integer ids using the compactify() function.

Faster Execution

For LDA the genism library provides a multicore implementation. To benefit from it, we use LdaMulticore() function with workers=#cores – 1.

**Doc2Vec**

We also generate vector representations of each question using the doc2vec [7], which is an extension of the word2vec embeddings learning provided by genism. We use the ‘distributed memory’ algorithm to learn 400 element vectors for each question in the corpus. We set the context window size to 10 words on each side of the predicted word and vary the learning rate ‘alpha’ from 0.025 to 0.001 over 50 iterations. Also, like in the case of LDA, we set workers = #cores -1 for parallel processing to speed up the learning process.

**Transformed Dataset Dimensions**

Using the LDA/LSI and Doc2Vec representations of the questions, the original dataset can be transformed into another dataset where the text of each question is replaced by 300 or 400 floating point numbers. The resulting width of the transformed datasets are 600 and 800 for LDA/LSI and Doc2Vec representations respectively as each row consists of two questions.

**Exploratory Model Building**

To begin, we built exploratory models on the initial 50-topic LDA representations using random forest and support vector machine classifiers. For our initial models with arbitrary hyperparameters the confusion matrices were as follows: -

**Random Forest (max depth = 20, number of estimators = 50 and max features = 20)**

*Confusion matrix:*

**Actual**

**Predicted**

|  |  |  |
| --- | --- | --- |
| **Class** | **Non-Duplicate** | **Duplicate** |
| **Non-Duplicate** | 81837 | 2138 |
| **Duplicate** | 42871 | 6570 |

**SVM**

*Confusion matrix:*

**Predicted**

**Actual**

|  |  |  |
| --- | --- | --- |
| **Class** | **Non-Duplicate** | **Duplicate** |
| **Non-Duplicate** | 83975 | 0 |
| **Duplicate** | 49441 | 0 |

We observed that the random forest classifier was able to learn some discriminatory features, whereas the SVM classifier totally failed. It is also worth mentioning that the time taken by the SVM was 2d 2h 42min as compared to only 1min 5sec for the random forest classifier. Based on these results, we decided not to use the SVM again because of its very long learning time.

**Final Model Building Process**

**Grappling with the Large Dataset**

We observed a lot of paging when we tried to apply 300-topic LDA/LSI representations to the dataset. Although we were able to get around it by applying these transformations in batches and saving the resulting vectors to file, the computer was unable to apply the regular classifiers to the final dataset having 600 columns.

To grapple with the situation, we were forced to use **out-of-core** classifiers from Sci-kit learn library that can operate on the datasets using incremental learning. These classifiers have a **partial\_fit()** function that can help in fitting the model incrementally to multiple batches. Out of such classifiers, we experimented with the SGD, Perceptron and Passive-Aggressive classifiers.

**Out-of-core Classifiers’ Performance**

To compare the performance of the available out-of-core classifiers, we used the 300-Topic LSI features.

Passive-Aggressive

Using the default configuration of the passive aggressive classifier, we achieved:

*Accuracy* = 64.3%

*Confusion Matrix*

**Actual**

**Predicted**

|  |  |  |
| --- | --- | --- |
| **Class** | **Non-Duplicate** | **Duplicate** |
| **Non-Duplicate** | 73053 | 10922 |
| **Duplicate** | 36660 | 12781 |

Perceptron

Using the default configuration of the perceptron classifier, we achieved:

*Accuracy* = 62.9%

*Confusion Matrix*

**Actual**

**Predicted**

|  |  |  |
| --- | --- | --- |
| **Class** | **Non-Duplicate** | **Duplicate** |
| **Non-Duplicate** | 69966 | 14009 |
| **Duplicate** | 35468 | 13973 |

SGD

Using the default configuration of the SGD classifier, we achieved a higher accuracy than the other classifiers. Therefore, we proceeded with **hyperparameter tuning** for this classifier. After using the best learnt parameters from the hyperparameter tuning (loss = ‘modified\_huber’, class\_weight=’balanced’), we got the following results. The other options evaluated were loss = ['hinge', 'log', 'squared\_hinge'] and class\_weight = None.

*Accuracy* = 67%

*Confusion Matrix:*

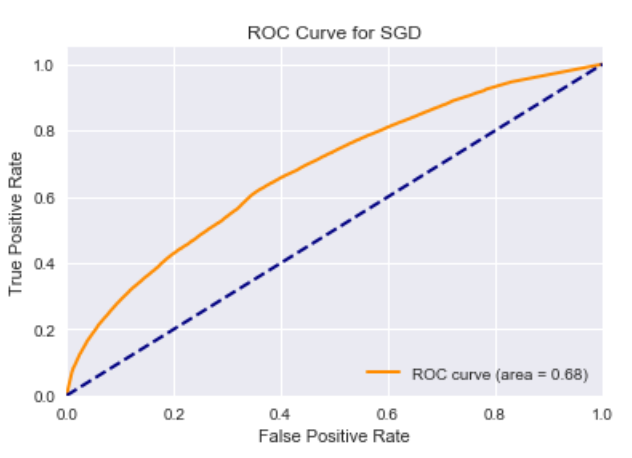
**Actual**

**Predicted**

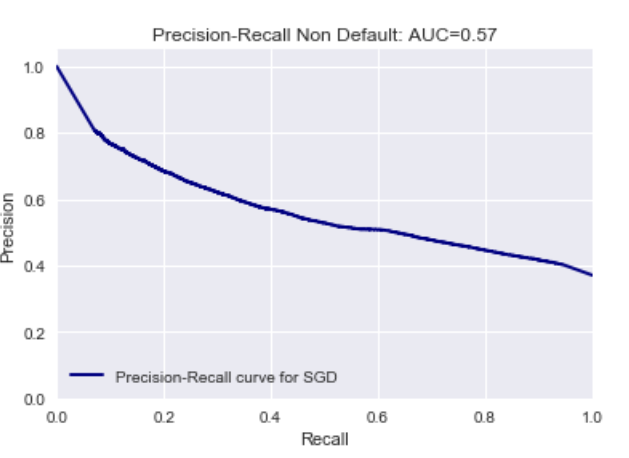
|  |  |  |
| --- | --- | --- |
| **Class** | **Non-Duplicate** | **Duplicate** |
| **Non-Duplicate** | 72518 | 11457 |
| **Duplicate** | 32530 | 16911 |

*ROC Curve:*

As may be seen, we got an area under the curve of 0.68 for the best hyperparameters of the SGD classifier.



*Precision Recall Curve:*



As, may be seen from the above curve there exists a clear tradeoff between the precision and recall. To achieve a high recall of about 95%, the precision drops to 40%, which is not very good.

**Using Deep Learning**

After having experimented with the given classifiers in the sci-kit learn library, we decided to try out neural networks. As we show in the next paragraphs, we achieved significant gains in accuracy and performance.

Neural Net Configuration when Learning from LSI/LDA Features

For training the neural networks, we used the Keras wrapper library with Theano as the backend. For faster learning we used the cuDNN library and a 640 core GPU.

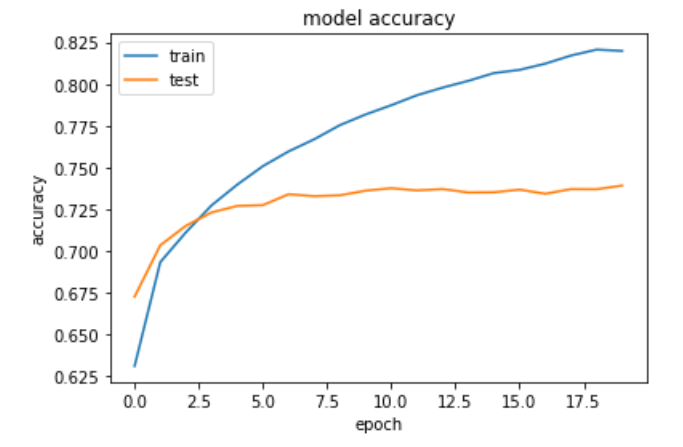
For the LSI/LDA features, we used the following configuration for our neural net.

* Input Layer: 600
* Hidden Layer1: 600 neurons, kernel\_initializer = ‘uniform’, activation=’relu’
* Hidden Layer2: 600 neurons, kernel\_initializer = ‘uniform’, activation=’relu’
* Output Layer: 1 neuron, kernel\_initializer = ‘uniform’, activation = ‘sigmoid’
* Loss function: ‘binary\_crossentropy’
* Optimizer: ‘adam’
* Epochs: 20
* Batch size: 10000

Performance with 300-Topic LSI Features

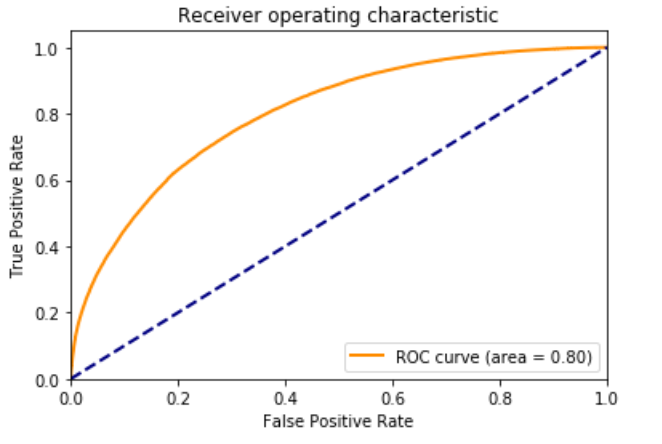
*Average 10-fold cross-validated accuracy*: 74.03 %

*Accuracy vs training epochs*



We observe that while the training set accuracy improves to about 82%, the accuracy over the validation set saturates close to a value of 73%.

*ROC curve for test set*

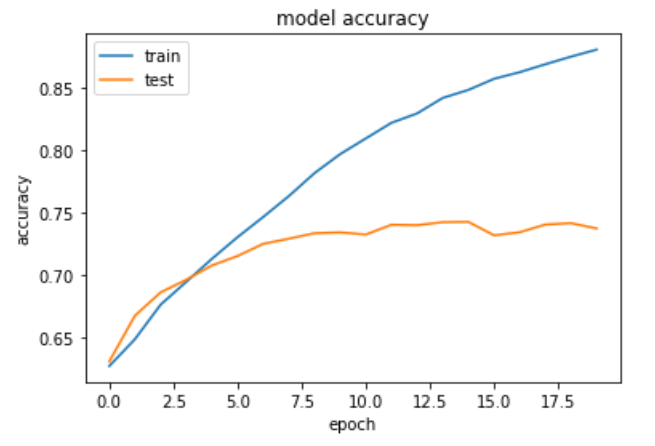


We observe that the area under the curve is 0.80 for the validation set.

Performance with 300-Topic LDA Features

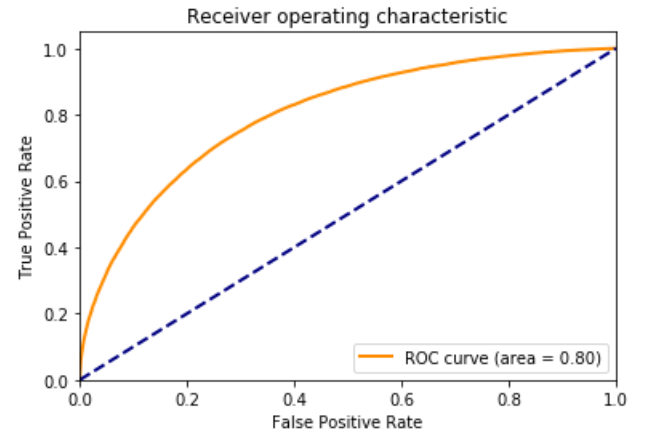
*Average 10-fold cross-validated accuracy*: 74.58 %, which is slightly better than the LSI features

*Accuracy vs training epochs*



We observe that the accuracy over the training set improves to about 87%, the accuracy over the validation set saturates to below 74%.

*ROC curve for test set*



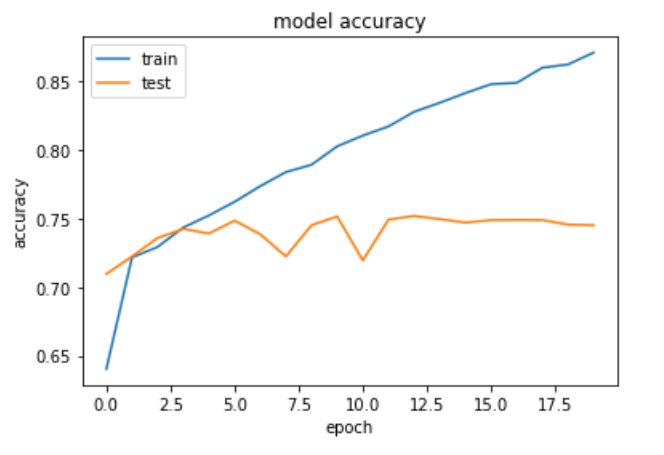
We see that the area under the ROC curve is the same as for LSI. We conclude that the LDA embedding is only marginally better than the LSI.

400-Dimensional Doc2Vec Features

For training on the 400-dimensions per question of the Doc2Vec features, we modify the number of neurons in the net to 800 for the input and hidden layers. The remaining configuration remains the same.

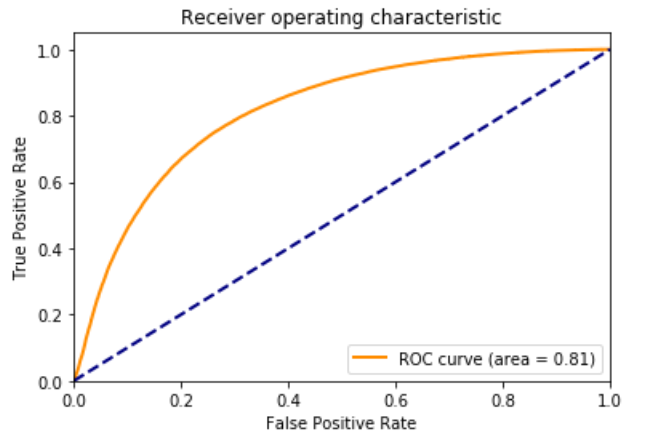
*Average 10-fold cross-validated accuracy*: 75.07%, which is better than both LSI and LDA

*Accuracy vs training epochs*



We observe that the accuracy over the training set improves to about 87%, the accuracy over the validation set saturates to about 75%.

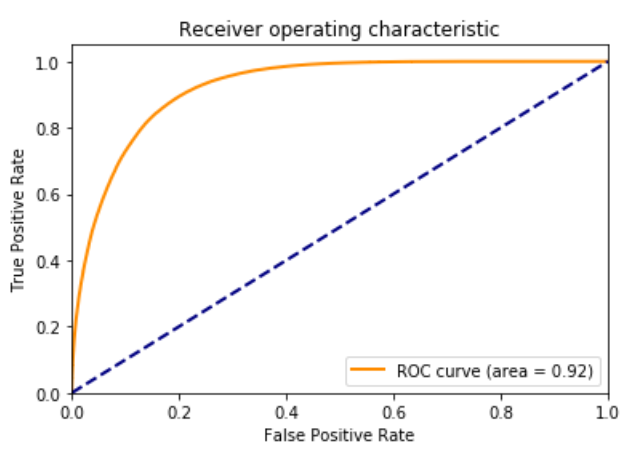
*ROC curve for test set*



Here we do see a 0.01 improvement in the area under the ROC curve as compared to the LSI/LDA features. We conclude that the word2vec embedding is a little better than the LSI/LDA.

*ROC curve for training set*

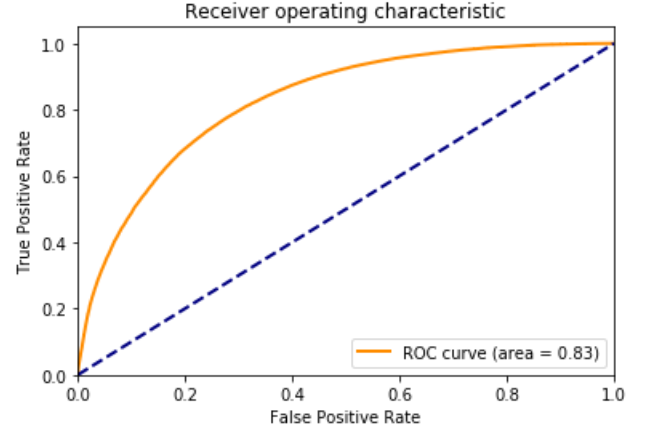
For the best performing model, we also look into the ROC curve over the training set.



We find that the area under the curve is 0.92 for the ROC curve plotted for the training set, which is indicative of overfitting. To reduce the overfitting, we try out adding a 20% drop out to the input and the hidden layers.

For the network with drop-out we plot both the test and training set ROC curves.

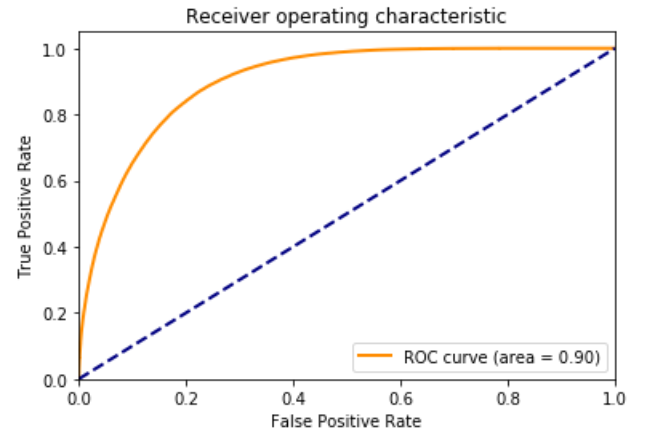
*ROC curve for test set*



Here we do see a 0.02 improvement to 0.83 in the area under the ROC curve as compared to the model without the drop out. This implies better generalization for the drop-out based network.

*ROC curve for training set*

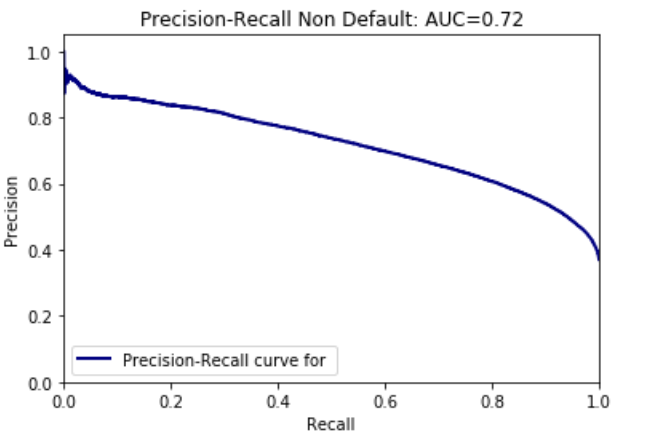
For the best performing model, we also look into the ROC curve over the training set.



We see a drop of 0.02 in the area under the ROC curve for the training set. This means that we have less overfitting when we employed the drop-out.

For our best performing model so far, we also plot the precision recall curve and look at the confusion matrix.

*Precision Recall Curve*



We observe that the precision-recall curve is not ideal but it shows a much better trade-off between precision and recall as compared to the non-neural network based classifiers.

*Confusion Matrix*

**Actual**

**Predicted**

|  |  |  |
| --- | --- | --- |
| **Class** | **Non-Duplicate** | **Duplicate** |
| **Non-Duplicate** | 62864 | 21342 |
| **Duplicate** | 12353 | 36857 |

**Conclusions**

In this project, we experimented with 3 different types of NLP feature generation namely Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA) and Doc2Vec embeddings to build a classification model that identifies questions with the same intent from the Quora questions pairs corpus.

We find that LDA is marginally better than the LSI, whereas Doc2Vec embedding works best in capturing the semantics of a question. We also find that the training time (no. of epochs) in computing the doc2vec embeddings does matter and the longer we train the better results we can achieve. In our project, we trained the doc2vec model for 10, 20 and 50 epochs. The results in the report are for the 50 epoch implementation. Other results were worse than the LDA/LSI.

We also experimented with three different out-of-core classifiers available in the Scikit-learn library. These are SGD, Perceptron and Passive-Aggressive classifier. All of these out-of-core classifiers have a partial\_fit() function that enables incremental training in batches over a large corpus. We find that the SGD outperforms the others for this task. We did hyperparameter tuning for SGD and found that the loss function of ‘modified\_huber’ works best for the problem.

However, we also find that none of these classifiers works as well as a neural network based classifier. By building a neural network model using Theano, we were able to achieve an accuracy improvement of at least 8 percentage points relative to the regular classifiers. The use of 20% dropout regularization marginally improves the classifier performance.

**Recommendations**

While we have achieved a reasonable level of accuracy by training a neural net over NLP features, we believe that the classifier performance can probably be improved by either training an LSTM neural net directly on the TfIdf model, where it may learn the features itself.

Ensembling the classifiers trained on multiple different features is also likely to further improve the prediction performance.

Statistical features may be included besides the NLP features to see what effect that can have over the classifier performance.

**References**

1. <https://www.kaggle.com/c/quora-question-pairs>
2. <https://spacy.io/>
3. <https://radimrehurek.com/gensim/>
4. <https://en.wikipedia.org/wiki/Tf%E2%80%93idf>
5. <https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation>
6. <https://nlp.stanford.edu/IR-book/html/htmledition/latent-semantic-indexing-1.html>