Machine learning engineer nanodegree

Capstone project

Duplicate question detection on QA Platforms

Ayman Fawzy Sherief | August 11th, 2020

I. Definition

a. Project overview

Since most data available on the web is text data, NLP has gained a lot of importance in the past decades as it stands to make out the most value out of the existing data.

Of the numerous problems that NLP can help solve, comes on important problem that may not be as popular as some of them, but it’s definitely not less important.

The problem is Question pairs classification, it’s a problem predominantly appearing on QA platforms such as yahoo answers, Quora, Stack Exchange, and more.

Simply put, people on QA platforms often ask similar questions in terms of what answers they’re seeking, but they do so in ways that may be syntactically different. Which means that several question could have exchangeable answers but be logically separated on a platform.

This creates two issues; one is that the user might be asking and waiting on an answer for a question that’s already answered somewhere else. The second is that with a large number of duplicate questions, the community could be strained to answer content that’s already been answered instead of contributing new content.

In an effort to help solve this issue. Quora, the QA platform, has contributed a labeled dataset of question pairs that is openly available for the community to try and test new methods of solving this problem.

In this project, we’ll go over this problem in detail, trying several solutions in the process and then deploying the best solution in a simple web app. Determining the best solution will be based on certain metrics that will be defined later on. But to be able to correctly define a metric for our model, we first need to know more about our data

b. Problem statement

To clearly state the problem, we'll be solving in this project: Given two text strings resembling two questions asked on a QA platform, output a label indication whether the two questions are essentially the same i.e. duplicate. Using 1 for duplicate questions and 0 for non-duplicate questions.

The resources we're going to use are the Quora question pairs dataset available on Kaggle, and we're going to explore it in the next few pages.

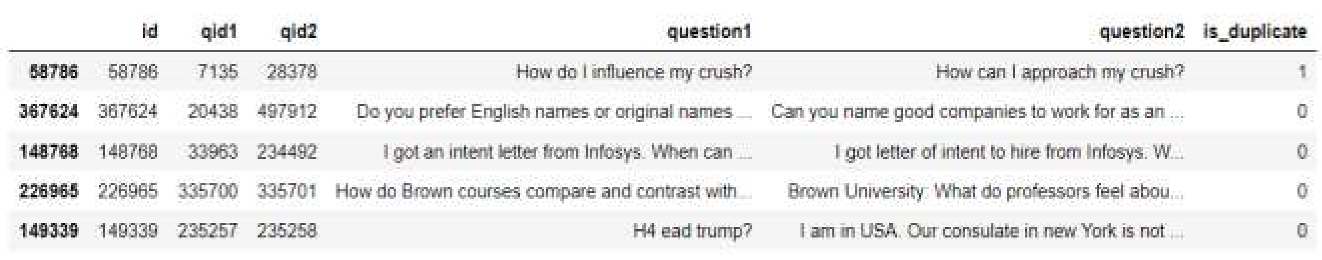
c. Metrics

The metrics we'll be using to evaluate our performance on this dataset are mainly: 1. Accuracy, 2. F1-score, And 3. Confusion matrix Since the dataset is unbalanced as we'll see in the analysis section. Accuracy alone might be misleading. It's certainly useful to use F1-score and take a look at the resulting confusion matrix to make sure our model isn't biased.

II. Analysis

a. Data Exploration

The Quora question pairs dataset comes in the form of a CSV formatted file. Here's a sample of the data.



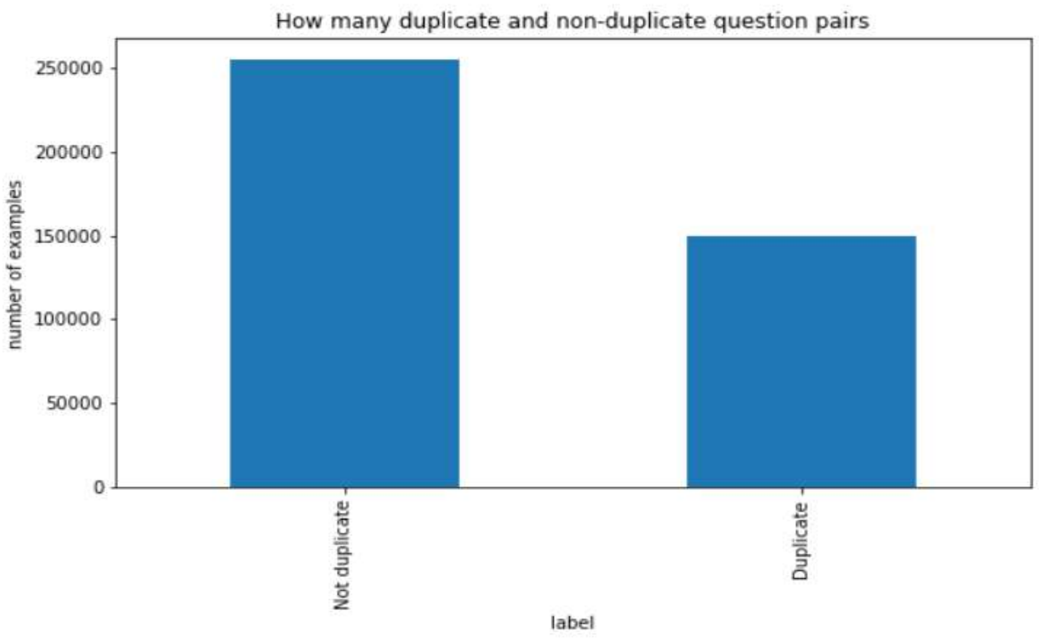
	id	qid1	qid2	question1	question2	is_duplicate
58786	58786	7135	28378	How do I influence my crush?	How can I approach my crush?	1
367624	367624	20438	497912	Do you prefer English names or original names ...	Can you name good companies to work for as an ...	0
148768	148768	33963	234492	I got an intent letter from Infosys. When can ...	I got letter of intent to hire from Infosys. W...	0
226965	226965	335700	335701	How do Brown courses compare and contrast with...	Brown University: What do professors feel abou...	0
149339	149339	235257	235258	H4 ead trump?	I am in USA. Our consulate in new York is not ...	0

The columns presented in this data are:

Id: represents the row id in the data
Qid1: represents the id for the first question
Qid2: represents the id for the second question
Question1: represent the text for the first question
Question2: represent the text for the second question
Is_duplicate: a label column, 1 if question 1 and 2 are duplicates, 0 otherwise.

b. Exploratory Visualization

Let’s take a look at how many duplicate vs non-duplicate examples we have in our dataset.



It appears that the data is unbalanced, to show exactly how much is that difference, using a few lines of code:

```
The data contains 404290 examples.
We have 149263 duplicate pairs and 255027 non-duplicate pairs
duplicate ratio is 0.37, more than one third of the data
```

The data is indeed unbalanced, which will be important to consider when training our models.

c. Algorithms and Techniques

Many NLP techniques have proven themselves in solving similar problems in the past. In this project I’m simply using algorithms that were already proven in the field and applying them to this problem.

To specifically mention the techniques I’m planning to use, there are mainly two techniques to preprocess the text data to be able to feed it into a model:

1. Vectorization: Which refers –in this context- to transforming the text from a string of characters into a vector of numbers, usually representing word counts or tf-idf values, this representation is traditionally used with classical models.
2. Tokenization: which refers -again in this context- to converting text form a sequence of words to a sequence of tokens (i.e. integer ids) that can be the later to create an embedding for these tokens and hence for the corresponding words. This technique is normally used with deep learning models.

I’ve mentioned several technical terms in the previous paragraph, if you didn’t get any of them, the next few paragraphs are for you.

The idea of vectorization is basically creating a large vector for every question we have, the dimension (i.e. length) of that vector is how many unique words we have in our entire dataset (or how many we choose to keep track of). So, in that vector, each word has its own index, the vector values are then filled with numbers representing how many times a word appears in that sentence.

There are some problems with that representation, The biggest one is useless but commonly used words such as words like (the, a, an, is, etc..) normally have big numbers that overestimate it’s importance in the context of the sentence, which is why we often use some other method to fill the vector values, namely Tf-Idf.

Tf-idf stands for term frequent – Inverse document frequency. It’s simply calculated by dividing the word count in that sentence by the log of it’s overall count in all the data for that specific word.

What that does is it gives less weight to words that appear a lot everywhere in the data, and more weight to words that appear in a small number of the examples in the data, since these mostly carry most of the distinction. Think of it as a way to normalize the word counts with regards to the entire dataset.

One other way to represent the data is tokenization. Which instead represents each word first by an integer ID. Then replaces this integer ID with a vector of some arbitrary dimension (typically in the order of 100 or so) such that words with similar meanings have similar vector representations. This has two obvious advantages over the previous representations, one is that it keeps the sequence of the words intact, meaning that two sentences with the same words In different order would have different representations (unlike the previous methods) and the second advantage being that different words with similar meanings would be represented similarly and the model -ideally- will be able to capture that similarity.

So, this was the preprocessing tools, But what about the models themselves?

Well, in this project. We'll use Several classical as well as deep learning models.

First, I'll be converting the data into the vectorized representation with Tf-idf, then stacking all features of both questions together. I'll then apply a simple Logistic regression model will be used, as a simple benchmark model. And then we'll try to fit an XGBoost classifier to the data. And try to achieve better accuracy than out benchmark model.

Second, we'll fit a simple neural network of a few stacked dense layers on the same vectorized representation of the data.

The last model to try will be a complex deep learning model. That consists of an embedding layer, an LSTM layer, followed by a dense layer. The three together work as a feature extractor from text and are run on both input questions, then the output of this network on both questions is concatenated and input into another dense layer that outputs a label. Hopefully the last layer will learn some similarity function.

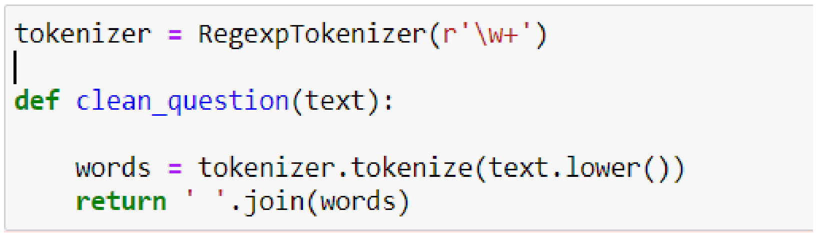
d. Benchmark

For this project, I've used two sources to benchmark my performance, the first is a blog post about the same problem that I'll include in the references, and the second is my own logistic regression model that I've mentioned above.

III. Methodology

a. Data preprocessing

In terms of data cleaning and preprocessing, I haven't used any complex steps, the code below is all the preprocessing that was needed.



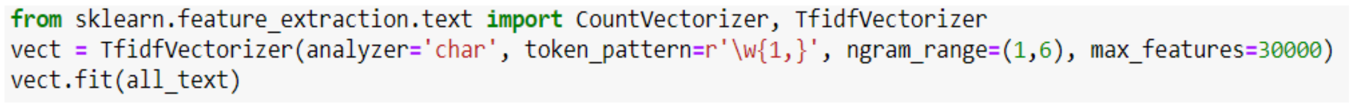
```
tokenizer = RegexpTokenizer(r'\w+')

def clean_question(text):

    words = tokenizer.tokenize(text.lower())
    return ' '.join(words)
```

By Applying this to both questions' text, I'm converting all text to lowercase and removing punctuation marks, making everything easier to handle in the model.

I've tried using stemming and lemmatization before I reached this final simple solution. But I've found that they hurt my accuracy when I attempted to use more complex feature representations and models.

b. <u>Implementation</u>

In this section, I'll discuss the implementation for the several lines of code In the Jupyter notebook that might need explaining



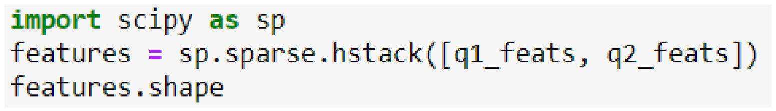
```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
vect = TfidfVectorizer(analyzer='char', token_pattern=r'\w{1,}', ngram_range=(1,6), max_features=30000)
vect.fit(all_text)
```

In this cell, I'm Creating the TF-idf vectorizer to be used later to convert my text data to tf-idf vectors.

What I'd like to point out is, I'm using a character-level analyzer, meaning my vectorizer will consider every character on its own, hence creating n-grams of characters, specifically 1-grams, 2-grams all the way up to 6-grams.

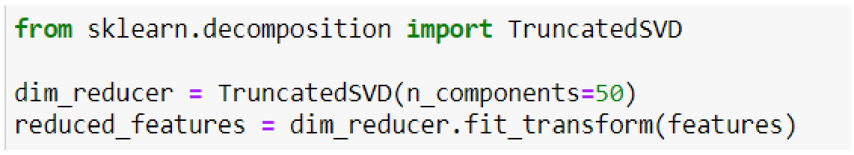
However, I'm not considering single characters, and I'm only looking at the top 30k features for each question.

Next, we have this cell:



```
import scipy as sp
features = sp.sparse.hstack([q1_feats, q2_feats])
features.shape
```

Here I'm simply stacking features for both questions together, so. For every question I have 30k features, and for every example I have two questions, meaning every training example in the features sparse matrix consists of 60k features from both question and a label.

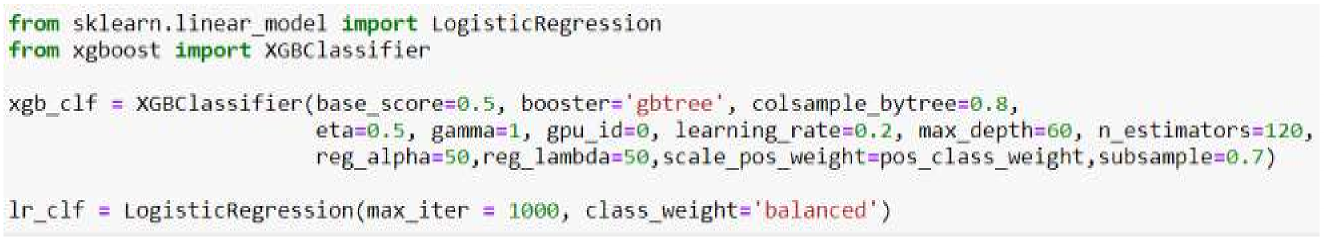


```
from sklearn.decomposition import TruncatedSVD

dim_reducer = TruncatedSVD(n_components=50)
reduced_features = dim_reducer.fit_transform(features)
```

In the above cell, I'm simply using Truncated SVD to create a dense representation of the data to use it later in training deep neural networks, since they can't handle large sparse data. And because of memory limitations on my machine.

```
class_counts = data.is_duplicate.value_counts()
pos_class_weight = class_counts[0] / class_counts[1]
```

Then as a final preparation step we calculate the weight for the positive (i.e. duplicate) class, to then use it in training our model to counter the imbalance in the data.



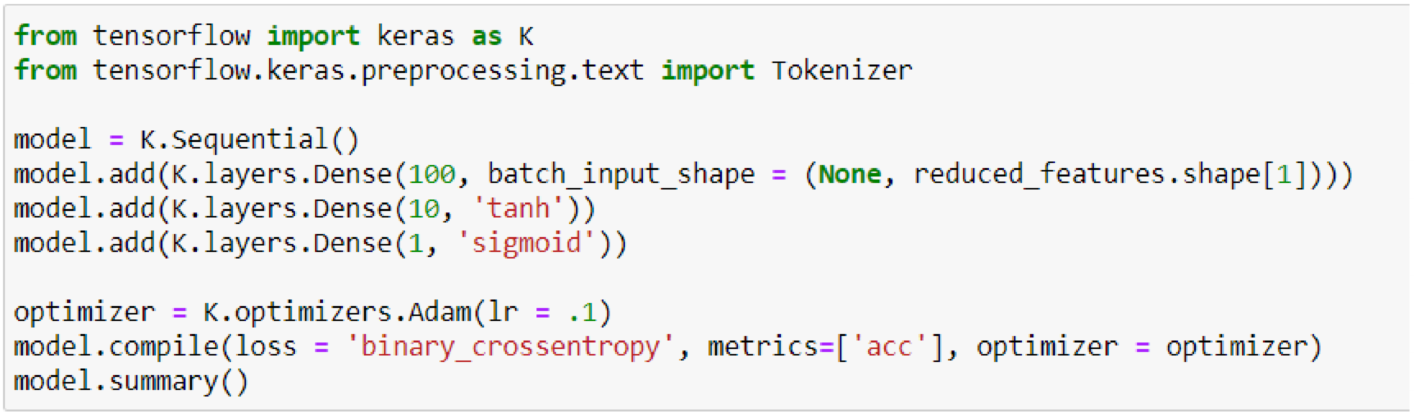
```
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier

xgb_clf = XGBClassifier(base_score=0.5, booster='gbtree', colsample_bytree=0.8,
                        eta=0.5, gamma=1, gpu_id=0, learning_rate=0.2, max_depth=60, n_estimators=120,
                        reg_alpha=50,reg_lambda=50,scale_pos_weight=pos_class_weight,subsample=0.7)

lr_clf = LogisticRegression(max_iter = 1000, class_weight='balanced')
```

Then, Comes the model’s implementation.

In the above cell, I created two models, one is a simple logistic regression model to use as a benchmark model, and the other is the XGboost model.

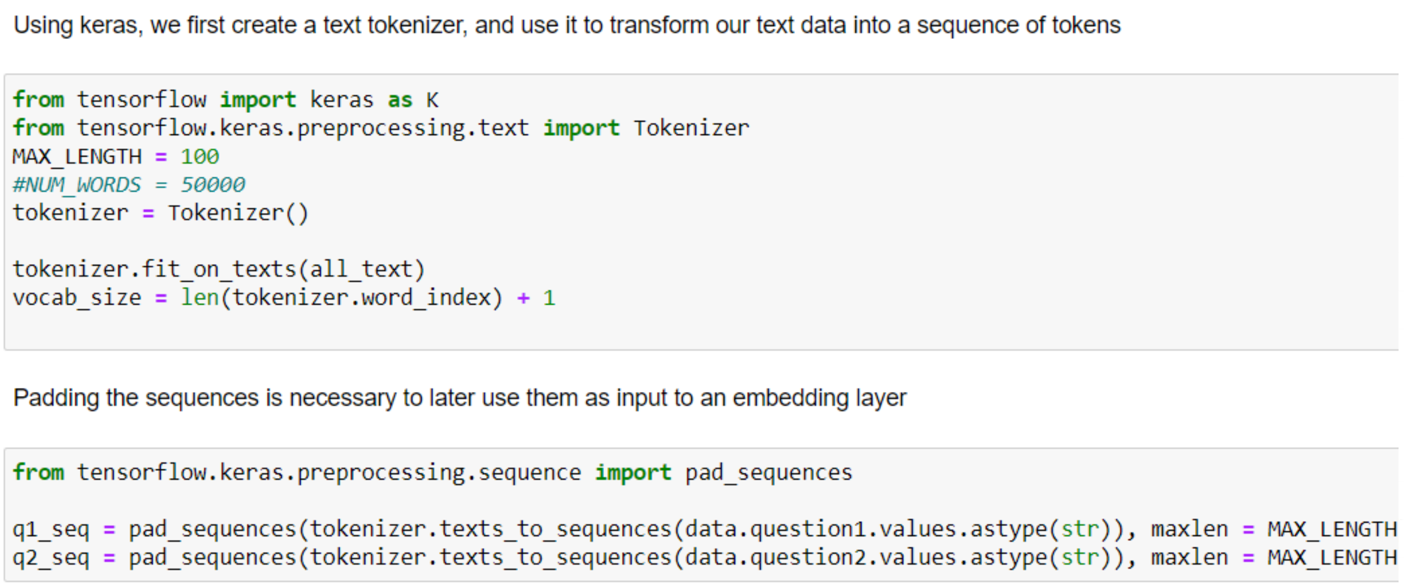
Please note that the specific hyperparameter values were acquired through a long and painful process of experimenting and searching. Basically, moving from an under-fitting problem by gradually increasing the model depth and number of estimators, to then an over-fitting problem that was later solved by increasing regularization values.

There are multiple cells testing and validation both these models. But I’ll be including them in the results section.



```
from tensorflow import keras as K
from tensorflow.keras.preprocessing.text import Tokenizer

model = K.Sequential()
model.add(K.layers.Dense(100, batch_input_shape = (None, reduced_features.shape[1])))
model.add(K.layers.Dense(10, 'tanh'))
model.add(K.layers.Dense(1, 'sigmoid'))

optimizer = K.optimizers.Adam(lr = .1)
model.compile(loss = 'binary_crossentropy', metrics=['acc'], optimizer = optimizer)
model.summary()
```

Next, we’ll be building our first, simple Deep neural network, training it on the reduced features of the Truncated SVD.



Using keras, we first create a text tokenizer, and use it to transform our text data into a sequence of tokens

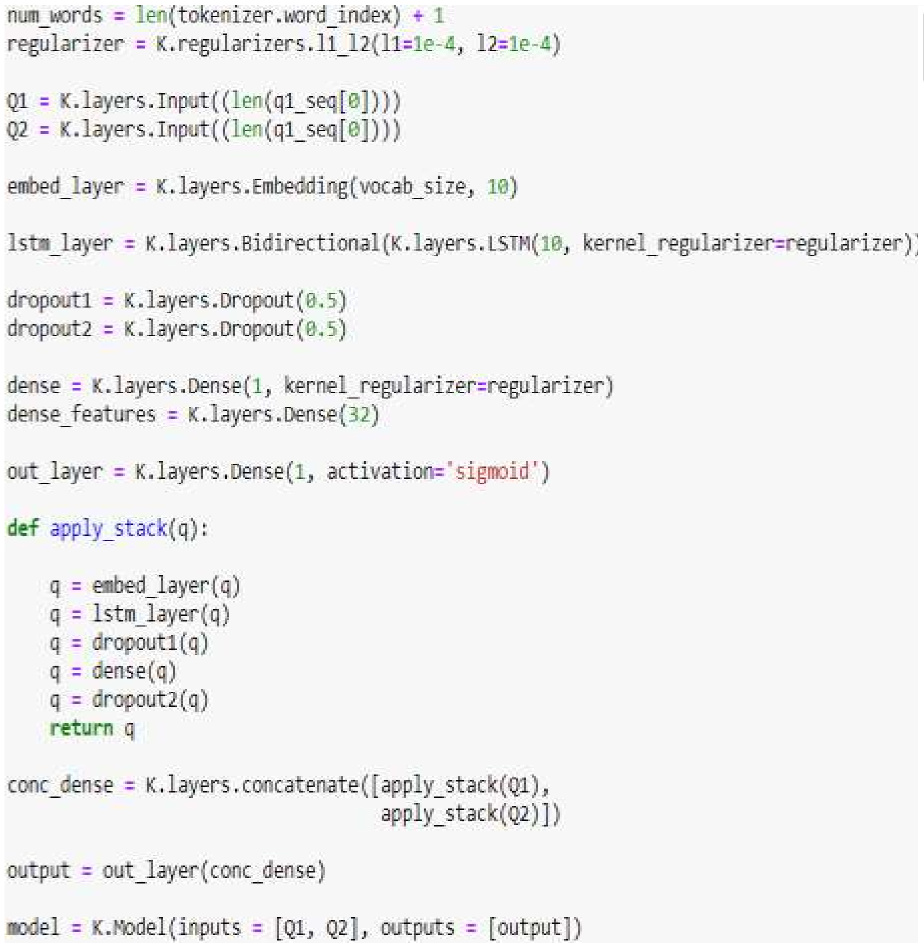
```
from tensorflow import keras as K
from tensorflow.keras.preprocessing.text import Tokenizer
MAX_LENGTH = 100
#NUM_WORDS = 50000
tokenizer = Tokenizer()

tokenizer.fit_on_texts(all_text)
vocab_size = len(tokenizer.word_index) + 1
```

Padding the sequences is necessary to later use them as input to an embedding layer

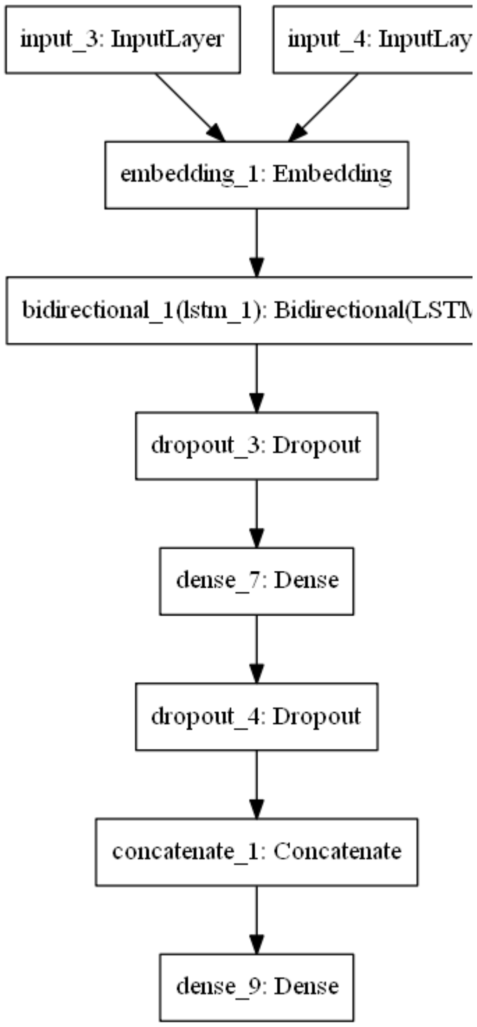
```
from tensorflow.keras.preprocessing.sequence import pad_sequences

q1_seq = pad_sequences(tokenizer.texts_to_sequences(data.question1.values.astype(str)), maxlen = MAX_LENGTH
q2_seq = pad_sequences(tokenizer.texts_to_sequences(data.question2.values.astype(str)), maxlen = MAX_LENGTH
```

The next step after that would be to prepare the data to be fed into a recurrent deep neural network model. Using the handy Keras library, this is easily done by converting the text into sequences then padding them all to be of the same length.

Next, I’ll discuss the recurrent deep neural network model architecture.



```
num_words = len(tokenizer.word_index) + 1
regularizer = K.regularizers.l1_l2(l1=1e-4, l2=1e-4)

Q1 = K.layers.Input((len(q1_seq[0])))
Q2 = K.layers.Input((len(q1_seq[0])))

embed_layer = K.layers.Embedding(vocab_size, 10)

lstm_layer = K.layers.Bidirectional(K.layers.LSTM(10, kernel_regularizer=regularizer))

dropout1 = K.layers.Dropout(0.5)
dropout2 = K.layers.Dropout(0.5)

dense = K.layers.Dense(1, kernel_regularizer=regularizer)
dense_features = K.layers.Dense(32)

out_layer = K.layers.Dense(1, activation='sigmoid')

def apply_stack(q):

    q = embed_layer(q)
    q = lstm_layer(q)
    q = dropout1(q)
    q = dense(q)
    q = dropout2(q)
    return q

conc_dense = K.layers.concatenate([apply_stack(Q1),
                                   apply_stack(Q2)])

output = out_layer(conc_dense)

model = K.Model(inputs = [Q1, Q2], outputs = [output])
```



This is the code used to create the model; However, I've created a simple graph using keras that better represents this model and can be better used to explain it.

As shown in the graph, this code uses the Keras functional model API to create a model that works by first inputting the two questions into an embedding layer, followed by a bidirectional LSTM layer, followed by a dense layer. Those three layers work here as a feature extractor on text and are applied independently to both questions at first.

The dense layer outputs are then concatenated together, and fed to a final Dense layer to create an output label. The dropout layers are simple here for regularization and don't perform any calculations during testing so I haven't mentioned them.

c. Refinement

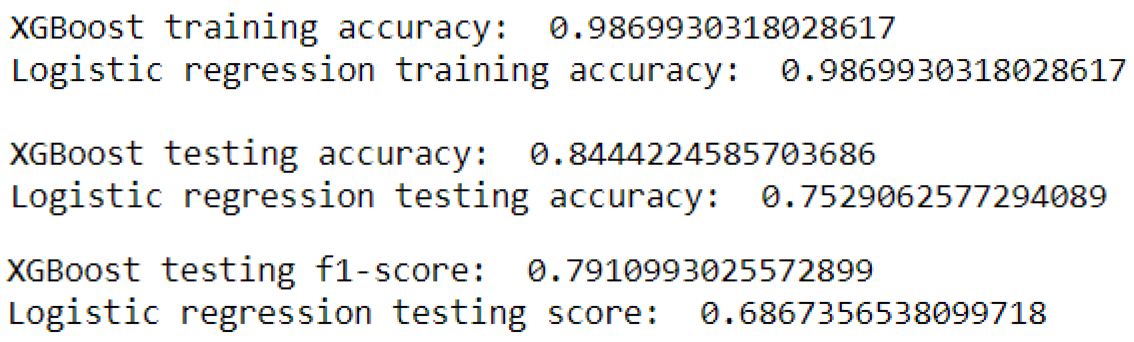
As I’ve mentioned before, I’ve already worked through several steps to improve my XGboost model by tuning it’s hyperparameters. However, a great deal of effort was also placed to modify this deep neural network architecture.

At first there was only one dense layer and a much larger LSTM. But then I found that adding an additional dense layer could help the model learn. And that for this problem, a bidirectional small LSTM performs better than a large regular one.

IV. Results

a. Model evaluation and validation

For all the models I’ve built and tested, I’ve split the data into a training and testing datasets. Taking into consideration the error on both when I made my decision. I’ve also calculated F1-scores for the classical models and plotted a confusion matrix.

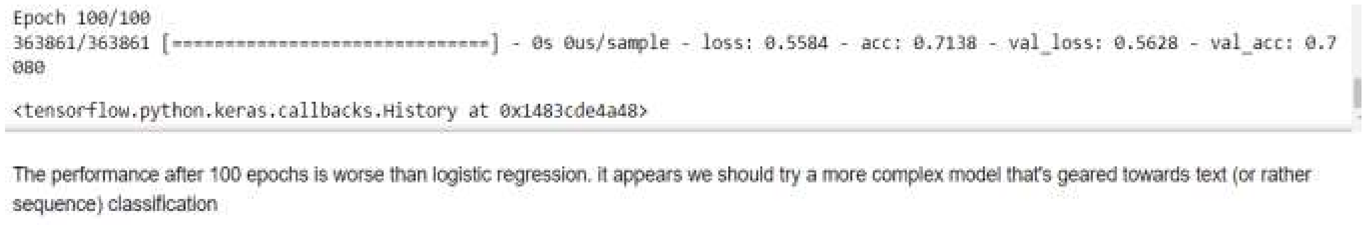


```
XGBoost training accuracy:  0.9869930318028617
Logistic regression training accuracy:  0.9869930318028617

XGBoost testing accuracy:  0.8444224585703686
Logistic regression testing accuracy:  0.7529062577294089

XGBoost testing f1-score:  0.7910993025572899
Logistic regression testing score:  0.6867356538099718
```

The above outputs and plots are from the notebook, and they briefly detail how my models did on the dataset. the take away here is:

1. The benchmark accuracy according to the logistic regression model is 75%
2. The XGBoost model did considerably better at 84%, with a good f1-score.

As for how the deep learning models performed.

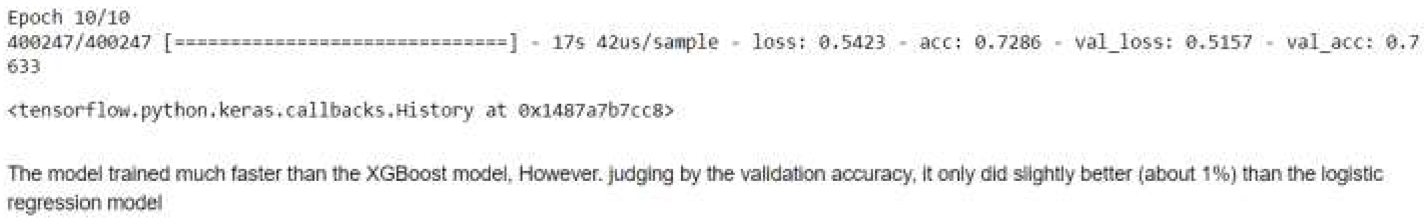


```
Epoch 100/100
363861/363861 [==============================] - 0s 0us/sample - loss: 0.5584 - acc: 0.7138 - val_loss: 0.5628 - val_acc: 0.7
080

<tensorflow.python.keras.callbacks.History at 0x1483cde4a48>
```

The performance after 100 epochs is worse than logistic regression. it appears we should try a more complex model that's geared towards text (or rather sequence) classification

We can see from this output snippet, that our deep simple deep learning neural net did considerably worse than the benchmark model, which means I need a more complex deep learning model.

As for the deep learning architecture I've presented above, it kept only doing slightly better than the logistic regression model. I've tried several different hyperparameters, optimizers, and architectures, but this seems to be the best I could get out of it.



```
Epoch 10/10
400247/400247 [==============================] - 17s 42us/sample - loss: 0.5423 - acc: 0.7286 - val_loss: 0.5157 - val_acc: 0.7
633

<tensorflow.python.keras.callbacks.History at 0x1487a7b7cc8>
```

The model trained much faster than the XGBoost model, However. judging by the validation accuracy, it only did slightly better (about 1%) than the logistic regression model

b. <u>Justification</u>

Judging by the large difference in accuracy between the deep learning architecture (76%) and the XGboost model (84%), The winner model is by far the XGBoost model, although it took much longer to train, which means that in a production environment maintaining it and updating it with new incoming data could be challenging. The accuracy it achieved accompanied by how balanced It is makes it the best model for this problem.

V. Conclusion

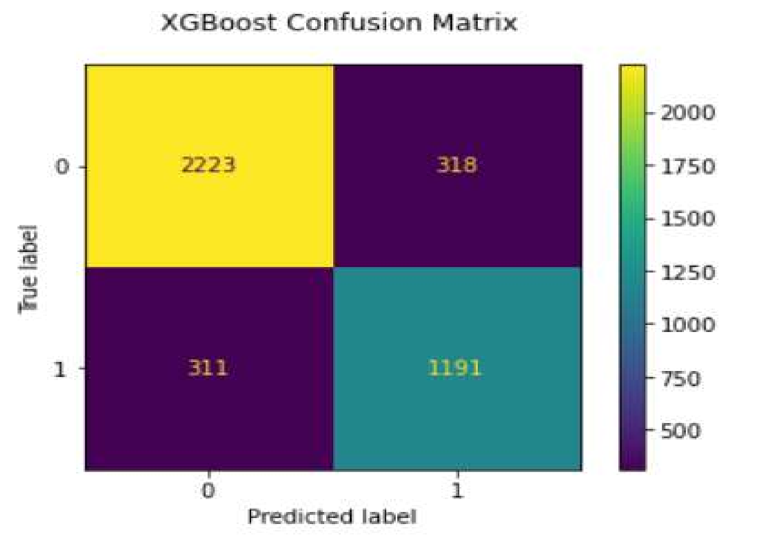
Once again, The XGBoost algorithm proves itself.

a. Free-form visualization

I'd like to include here the Confusion matrix of the XGBoost model

This Clearly shows that even though the data was unbalanced, the model is unbiased and the results can be trusted. Since the number of false positives and false negatives is similar.

```
XGBoost confusion matrix

Text(0.5, 0.98, 'XGBoost Confusion Matrix')
```



b. Reflection

In this report, I've shown how I have First explored the problem of question similarity in QA platforms, and then acquired open source data from Kaggle to attempt a solution to the problem. Then I've experimented with many ways to clean, preprocess, and transform the data to feed it into models, along with experimenting with different model hyperparameters and even different deep learning architectures. Monitoring and comparing results as I go. And as I reached an accuracy score, I couldn't improve much upon. I started moving from experimentation to making sure the model is robust and unbiased. And moved to deploying it in a simple web app.

c. Improvement

Further work on this project may include:

1- Coming up with better preprocessing that doesn't hurt performance.
2- Trying state-of-the-art deep learning models on the problem (transformers, attention mechanism, Siamese networks, and more)
3- Improving upon the deployment of the model

VI. References

- Kaggle's Quora Question Pairs Dataset
- Analytics Vidhya's medium post by Vedansh Sharma
- Scikit-learn Tf-Idf vectorizer documentation
- GloVe Word Embedding by Jeffrey Pennington, Richard Socher, and Christopher D. Manning