32 min read

This tutorial shows how to build an NLP project with TensorFlow that explicates the semantic similarity between sentences using the Quora dataset. It is based on the work of Abhishek Thakur, who originally developed a solution on the Keras package.

> *This article is an excerpt from a book written by Luca Massaron, Alberto Boschetti, Alexey* Grigorev*, Abhishek Thakur, and Rajalingappaa Shanmugamani titled TensorFlow Deep Learning Projects.*

Presenting the dataset

The data, made available for non-commercial purposes (https://www.quora.com/about/tos) in a Kaggle competition (https://www.kaggle.com/c/quora-question-pairs) and on Quora's blog (https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs), consists of 404,351 question pairs with 255,045 negative samples (non-duplicates) and 149,306 positive samples (duplicates). There are approximately 40% positive samples, a slight imbalance that won't need particular corrections.

Actually, as reported on the Quora blog, given their original sampling strategy, the number of duplicated examples in the dataset was much higher than the non-duplicated ones. In order to set up a more balanced dataset, the negative examples were upsampled by using pairs of related questions, that is, questions about the same topic that are actually not similar.

Before starting work on this project, you can simply directly download the data, which is about 55 MB, from its Amazon S3 repository at this link into our working directory.

After loading it, we can start diving directly into the data by picking some example rows and examining them. The following diagram shows an actual snapshot of the few first rows from the dataset:

	id	qid1	qid2	question1	question2	is_duplicate
0	0	1	2	What is the step by step guide to invest in sh...	What is the step by step guide to invest in sh...	0
1	1	3	4	What is the story of Kohinoor (Koh-i-Noor) Dia...	What would happen if the Indian government sto...	0
2	2	5	6	How can I increase the speed of my internet co...	How can Internet speed be increased by hacking...	0
3	3	7	8	Why am I mentally very lonely? How can I solve...	Find the remainder when [math]23^{24}[/math] i...	0
4	4	9	10	Which one dissolve in water quikly sugar, salt...	Which fish would survive in salt water?	0

Exploring further into the data, we can find some examples of question pairs that mean the same thing, that is, duplicates, as follows:

How does Quora quickly mark questions as needing improvement?	Why does Quora mark my questions as needing improvement/clarification before I have time to give it details? Literally within seconds…
Why did Trump win the Presidency?	How did Donald Trump win the 2016 Presidential Election?
What practical applications might evolve from the discovery of the Higgs Boson?	What are some practical benefits of the discovery of the Higgs Boson?

At first sight, duplicated questions have quite a few words in common, but they could be very different in length.

On the other hand, examples of non-duplicate questions are as follows:

Who should I address my cover letter to if I'm applying to a big company like Mozilla?	Which car is better from a safety persepctive? swift or grand i10. My first priority is safety?
Mr. Robot (TV series): Is Mr. Robot a good representation of real-life hacking and hacking culture? Is the depiction of hacker societies realistic?	What mistakes are made when depicting hacking in Mr. Robot compared to real-life cyber security breaches or just a regular use of technologies?
How can I start an online shopping (e-commerce) website?	Which web technology is best suited for building a big e-commerce website?

Some questions from these examples are clearly not duplicated and have few words in common, but some others are more difficult to detect as unrelated. For instance, the second pair in the example might turn to be appealing to some and leave even a human judge uncertain. The two questions might mean different things: *why* versus *how*, or they could be intended as the same from a superficial examination.

Looking deeper, we may even find more doubtful examples and even some clear data mistakes; we surely have some anomalies in the dataset (as the Quota post on the dataset warned) but, given that the data is

derived from a real-world problem, we can't do anything but deal with this kind of imperfection and strive to find a robust solution that works.

At this point, our exploration becomes more quantitative than qualitative and some statistics on the question pairs are provided here:

Average number of characters in question1	59.57
Minimum number of characters in question1	1
Maximum number of characters in question1	623
Average number of characters in question2	60.14
Minimum number of characters in question2	1
Maximum number of characters in question2	1169

Question 1 and question 2 are roughly the same average characters, though we have more extremes in question 2. There also must be some trash in the data, since we cannot figure out a question made up of a single character.

We can even get a completely different vision of our data by plotting it into a word cloud and highlighting the most common words present in the dataset:

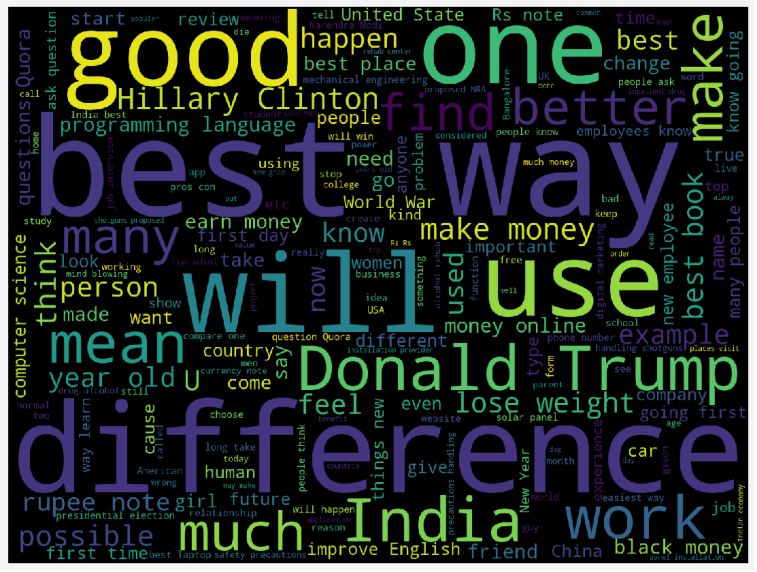


Figure 1: A word cloud made up of the most frequent words to be found in the Quora dataset
The presence of word sequences such as Hillary Clinton and Donald Trump reminds us that the data was gathered at a certain historical moment and that many questions we can find inside it are clearly ephemeral, reasonable only at the very time the dataset was collected. Other topics, such as programming language, World War, or earn money could be longer lasting, both in terms of interest and in the validity of the answers provided.

After exploring the data a bit, it is now time to decide what target metric we will strive to optimize in our project. Throughout the article, we will be using accuracy as a metric to evaluate the performance of our models. Accuracy as a measure is simply focused on the effectiveness of the prediction, and it may miss some important differences between alternative models, such as discrimination power (is the model more able to detect duplicates or not?) or the exactness of probability scores (how much margin is there between being a duplicate and not being one?).

We chose accuracy based on the fact that this metric was the one decided on by Quora's engineering team to create a benchmark for this dataset (as stated in this blog post of theirs: https://engineering.quora.com/Semantic-Question-Matching-with-Deep-Learning). Using accuracy as the metric makes it easier for us to evaluate and compare our models with the one from Quora's engineering team, and also several other research papers. In addition, in a real-world application, our work may simply be evaluated on the basis of how many times it is just right or wrong, regardless of other considerations.

We can now proceed furthermore in our projects with some very basic feature engineering to start with.

Starting with basic feature engineering

Before starting to code, we have to load the dataset in Python and also provide Python with all the necessary packages for our project. We will need to have these packages installed on our system (the latest versions should suffice, no need for any specific package version):

- `Numpy`
- `pandas`
- `fuzzywuzzy`
- `python-Levenshtein`
- `scikit-learn`
- `gensim`
- `pyemd`
- `NLTK`

As we will be using each one of these packages in the project, we will provide specific instructions and tips to install them.

For all dataset operations, we will be using pandas (and Numpy will come in handy, too). To install `numpy` and `pandas`:

```
pip install numpy
pip install pandas
```

The dataset can be loaded into memory easily by using pandas and a specialized data structure, the pandas dataframe (we expect the dataset to be in the same directory as your script or Jupyter notebook):

```
import pandas as pd
import numpy as np
data = pd.read_csv('quora_duplicate_questions.tsv', sep='t')
data = data.drop(['id', 'qid1', 'qid2'], axis=1)
```

We will be using the pandas dataframe denoted by `data` , and also when we work with our TensorFlow model and provide input to it.

We can now start by creating some very basic features. These basic features include length-based features and string-based features:

1. Length of question1
2. Length of question2
3. Difference between the two lengths
4. Character length of question1 without spaces
5. Character length of question2 without spaces
6. Number of words in question1
7. Number of words in question2
8. Number of common words in question1 and question2

These features are dealt with one-liners transforming the original input using the pandas package in Python and its method `apply`:

```
# length based features
data['len_q1'] = data.question1.apply(lambda x: len(str(x)))
data['len_q2'] = data.question2.apply(lambda x: len(str(x)))
# difference in lengths of two questions
data['diff_len'] = data.len_q1 - data.len_q2

# character length based features
data['len_char_q1'] = data.question1.apply(lambda x:
len(''.join(set(str(x).replace(' ', '')))))
data['len_char_q2'] = data.question2.apply(lambda x:
len(''.join(set(str(x).replace(' ', '')))))

# word length based features
data['len_word_q1'] = data.question1.apply(lambda x:
len(str(x).split()))
data['len_word_q2'] = data.question2.apply(lambda x:
len(str(x).split()))

# common words in the two questions
data['common_words'] = data.apply(lambda x:
len(set(str(x['question1'])
.lower().split())
.intersection(set(str(x['question2'])
.lower().split()))), axis=1)
```

For future reference, we will mark this set of features as feature set-1 or `fs_1`:

```
fs_1 = ['len_q1', 'len_q2', 'diff_len', 'len_char_q1',
        'len_char_q2', 'len_word_q1', 'len_word_q2',
        'common_words']
```

This simple approach will help you to easily recall and combine a different set of features in the machine learning models we are going to build, turning comparing different models run by different feature sets into a piece of cake.

Creating fuzzy features

The next set of features are based on fuzzy string matching. Fuzzy string matching is also known as approximate string matching and is the process of finding strings that approximately match a given pattern. The closeness of a match is defined by the number of primitive operations necessary to convert the string into an exact match. These primitive operations include insertion (to insert a character at a given position), deletion (to delete a particular character), and substitution (to replace a character with a new one).

Fuzzy string matching is typically used for spell checking, plagiarism detection, DNA sequence matching, spam filtering, and so on and it is part of the larger family of edit distances, distances based on the idea that a string can be transformed into another one. It is frequently used in natural language processing and other applications in order to ascertain the grade of difference between two strings of characters.

It is also known as Levenshtein distance, from the name of the Russian scientist, Vladimir Levenshtein, who introduced it in 1965.

These features were created using the `fuzzywuzzy` package available for Python (https://pypi.python.org/pypi/fuzzywuzzy). This package uses Levenshtein distance to calculate the differences in two sequences, which in our case are the pair of questions.

The `fuzzywuzzy` package can be installed using pip3:

```
pip install fuzzywuzzy
```

As an important dependency, `fuzzywuzzy` requires the `Python-Levenshtein` package (https://github.com/ztane/python-Levenshtein/), which is a blazingly fast implementation of this classic algorithm, powered by compiled C code. To make the calculations much faster using `fuzzywuzzy`, we also need to install the `Python-Levenshtein` package:

```
pip install python-Levenshtein
```

The `fuzzywuzzy` package offers many different types of ratio, but we will be using only the following:

1. QRatio
2. WRatio
3. Partial ratio
4. Partial token set ratio
5. Partial token sort ratio
6. Token set ratio
7. Token sort ratio

Examples of `fuzzywuzzy` features on Quora data:

```
from fuzzywuzzy import fuzz
```

fuzz.QRatio("Why did Trump win the Presidency?",
"How did Donald Trump win the 2016 Presidential Election")

This code snippet will result in the value of 67 being returned:

```
fuzz.QRatio("How can I start an online shopping (e-commerce) website?", "Which web technology is bes
```

In this comparison, the returned value will be 60. Given these examples, we notice that although the values of `QRatio` are close to each other, the value for the similar question pair from the dataset is higher than the pair with no similarity. Let's take a look at another feature from fuzzywuzzy for these same pairs of questions:

```
fuzz.partial_ratio("Why did Trump win the Presidency?",
"How did Donald Trump win the 2016 Presidential Election")
```

In this case, the returned value is 73:

```
fuzz.partial_ratio("How can I start an online shopping (e-commerce) website?", "Which web technology
```

Now the returned value is 57.

Using the `partial_ratio` method, we can observe how the difference in scores for these two pairs of questions increases notably, allowing an easier discrimination between being a duplicate pair or not. We assume that these features might add value to our models.

By using pandas and the `fuzzywuzzy` package in Python, we can again apply these features as simple one-liners:

```
    data['fuzz_qratio'] = data.apply(lambda x: fuzz.QRatio(
        str(x['question1']), str(x['question2'])), axis=1)
    data['fuzz_WRatio'] = data.apply(lambda x: fuzz.WRatio(
```

```
        str(x['question1']), str(x['question2'])), axis=1)

        data['fuzz_partial_ratio'] = data.apply(lambda x:
        fuzz.partial_ratio(str(x['question1']),
        str(x['question2'])), axis=1)

        data['fuzz_partial_token_set_ratio'] = data.apply(lambda x:
        fuzz.partial_token_set_ratio(str(x['question1']),
        str(x['question2'])), axis=1)

        data['fuzz_partial_token_sort_ratio'] = data.apply(lambda x:
        fuzz.partial_token_sort_ratio(str(x['question1']),
        str(x['question2'])), axis=1)

        data['fuzz_token_set_ratio'] = data.apply(lambda x:
        fuzz.token_set_ratio(str(x['question1']),
        str(x['question2'])), axis=1)
```

```
data['fuzz_token_sort_ratio'] = data.apply(lambda x:
                    fuzz.token_sort_ratio(str(x['question1']),
                    str(x['question2'])), axis=1)
```

This set of features are henceforth denoted as feature set-2 or `fs_2`:

```
fs_2 = ['fuzz_qratio', 'fuzz_WRatio', 'fuzz_partial_ratio',
       'fuzz_partial_token_set_ratio', 'fuzz_partial_token_sort_ratio',
       'fuzz_token_set_ratio', 'fuzz_token_sort_ratio']
```

Again, we will store our work and save it for later use when modeling.

Resorting to TF-IDF and SVD features

The next few sets of features are based on TF-IDF and SVD. **Term Frequency-Inverse Document Frequency** (**TF-IDF**). Is one of the algorithms at the foundation of information retrieval. Here, the algorithm is explained using a formula:

$$TF(t) = C(t)/N$$

$$IDF(t) = log(ND/ND_t)$$

You can understand the formula using this notation: *C(t)* is the number of times a term *t* appears in a document, *N* is the total number of terms in the document, this results in the **Term Frequency** (**TF**). ND is the total number of documents and ND_t is the number of documents containing the term *t*, this provides the **Inverse Document Frequency** (**IDF**). TF-IDF for a term *t* is a multiplication of Term Frequency and Inverse Document Frequency for the given term *t*:

$$TFIDF(t) = TF(t) * IDF(t)$$

Without any prior knowledge, other than about the documents themselves, such a score will highlight all the terms that could easily discriminate a document from the others, down-weighting the common words that won't tell you much, such as the common parts of speech (such as articles, for instance).

If you need a more hands-on explanation of TFIDF, this great online tutorial will help you try coding the algorithm yourself and testing it on some text data: https://stevenloria.com/tf-idf/
For convenience and speed of execution, we resorted to the `scikit-learn` implementation of TFIDF. If you don't already have `scikit-learn` installed, you can install it using pip:

```
pip install -U scikit-learn
```

We create TFIDF features for both question1 and question2 separately (in order to type less, we just deep copy the question1 `TfidfVectorizer`):

```
from sklearn.feature_extraction.text import TfidfVectorizer
from copy import deepcopy
tfv_q1 = TfidfVectorizer(min_df=3,
max_features=None,
strip_accents='unicode',
analyzer='word',
token_pattern=r'w{1,}',
ngram_range=(1, 2),
use_idf=1,
smooth_idf=1,
sublinear_tf=1,
stop_words='english')

tfv_q2 = deepcopy(tfv_q1)
```

It must be noted that the parameters shown here have been selected after quite a lot of experiments. These parameters generally work pretty well with all other problems concerning natural language processing, specifically text classification. One might need to change the stop word list to the language in question.

We can now obtain the TFIDF matrices for question1 and question2 separately:

```
q1_tfidf = tfv_q1.fit_transform(data.question1.fillna(""))
q2_tfidf = tfv_q2.fit_transform(data.question2.fillna(""))
```

In our TFIDF processing, we computed the TFIDF matrices based on all the data available (we used the `fit_transform` method). This is quite a common approach in Kaggle competitions because it helps to score higher on the leaderboard. However, if you are working in a real setting, you may want to exclude a part of the data as a training or validation set in order to be sure that your TFIDF processing helps your model to generalize to a new, unseen dataset.
After we have the TFIDF features, we move to SVD features. SVD is a feature decomposition method and it stands for singular value decomposition. It is largely used in NLP because of a technique called Latent Semantic Analysis (LSA).

A detailed discussion of SVD and LSA is beyond the scope of this article, but you can get an idea of their workings by trying these two approachable and clear online tutorials: https://alyssaq.github.io/2015/singular-value-decomposition-visualisation/ and https://technowiki.wordpress.com/2011/08/27/latent-semantic-analysis-lsa-tutorial/
To create the SVD features, we again use `scikit-learn` implementation. This implementation is a variation of traditional SVD and is known as `TruncatedSVD`.

A `TruncatedSVD` is an approximate SVD method that can provide you with reliable yet computationally fast SVD matrix decomposition. You can find more hints about how this technique works and it can be applied by consulting this web page: http://langvillea.people.cofc.edu/DISSECTION-LAB/Emmie'sLSI-SVDModule/p5module.html

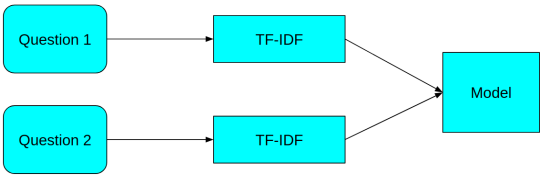
```
from sklearn.decomposition import TruncatedSVD
svd_q1 = TruncatedSVD(n_components=180)
svd_q2 = TruncatedSVD(n_components=180)
```

We chose 180 components for SVD decomposition and these features are calculated on a TF-IDF matrix:

```
question1_vectors = svd_q1.fit_transform(q1_tfidf)
question2_vectors = svd_q2.fit_transform(q2_tfidf)
```

Feature set-3 is derived from a combination of these TF-IDF and SVD features. For example, we can have only the TF-IDF features for the two questions separately going into the model, or we can have the TF-IDF of the two questions combined with an SVD on top of them, and then the model kicks in, and so on. These features are explained as follows.

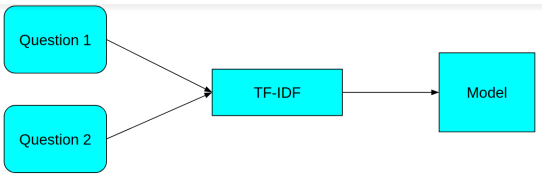
Feature set-3(1) or `fs3_1` is created using two different TF-IDFs for the two questions, which are then stacked together horizontally and passed to a machine learning model:



This can be coded as:

```
    from scipy import sparse
    # obtain features by stacking the sparse matrices together
    fs3_1 = sparse.hstack((q1_tfidf, q2_tfidf))
```

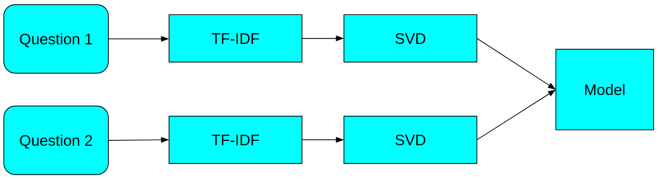
Feature set-3(2), or `fs3_2`, is created by combining the two questions and using a single TF-IDF:




```
tfv = TfidfVectorizer(min_df=3,
                      max_features=None,
                      strip_accents='unicode',
                      analyzer='word',
                      token_pattern=r'w{1,}',
                      ngram_range=(1, 2),
                      use_idf=1,
                      smooth_idf=1,
                      sublinear_tf=1,
                      stop_words='english')
```

```
    # combine questions and calculate tf-idf
    q1q2 = data.question1.fillna("")
    q1q2 += " " + data.question2.fillna("")
    fs3_2 = tfv.fit_transform(q1q2)
```

The next subset of features in this feature set, feature set-3(3) or `fs3_3`, consists of separate TF-IDFs and SVDs for both questions:

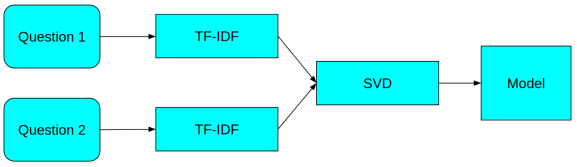


This can be coded as follows:

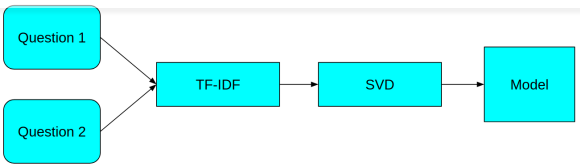
```
# obtain features by stacking the matrices together
fs3_3 = np.hstack((question1_vectors, question2_vectors))
```

We can similarly create a couple more combinations using TF-IDF and SVD, and call them `fs3-4` and `fs3-5`, respectively. These are depicted in the following diagrams, but the code is left as an exercise for the reader.

Feature set-3(4) or `fs3-4`:



Feature set-3(5) or `fs3-5`:



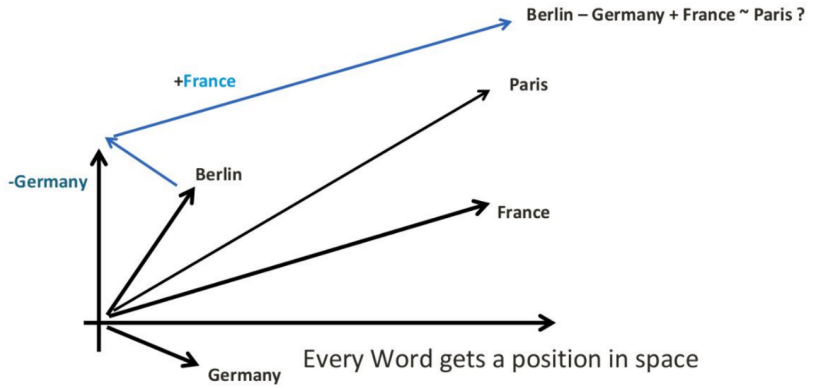
After the basic feature set and some TF-IDF and SVD features, we can now move to more complicated features before diving into the machine learning and deep learning models.

Mapping with Word2vec embeddings

Very broadly, Word2vec models are two-layer neural networks that take a text corpus as input and output a vector for every word in that corpus. After fitting, the words with similar meaning have their vectors close to each other, that is, the distance between them is small compared to the distance between the vectors for words that have very different meanings.

Nowadays, Word2vec has become a standard in natural language processing problems and often it provides very useful insights into information retrieval tasks. For this particular problem, we will be using the Google news vectors. This is a pretrained Word2vec model trained on the Google News corpus.

Every word, when represented by its Word2vec vector, gets a position in space, as depicted in the following diagram:



All the words in this example, such as Germany, Berlin, France, and Paris, can be represented by a 300-dimensional vector, if we are using the pretrained vectors from the Google news corpus. When we use Word2vec representations for these words and we subtract the vector of Germany from the vector of Berlin and add the vector of France to it, we will get a vector that is very similar to the vector of Paris. The Word2vec model thus carries the meaning of words in the vectors. The information carried by these vectors constitutes a very useful feature for our task.

For a user-friendly, yet more in-depth, explanation and description of possible applications of Word2vec, we suggest reading https://www.distilled.net/resources/a-beginners-guide-to-Word2vec-aka-whats-the-opposite-of-canada/, or if you need a more mathematically defined explanation, we recommend reading this paper: http://www.1-4-5.net/~dmm/ml/how_does_Word2vec_work.pdf

To load the Word2vec features, we will be using Gensim. If you don't have Gensim, you can install it easily using pip. At this time, it is suggested you also install the pyemd package, which will be used by the WMD distance function, a function that will help us to relate two Word2vec vectors:

```
pip install gensim
pip install pyemd
```

To load the Word2vec model, we download the `GoogleNews-vectors-negative300.bin.gz` binary and use Gensim's `load_Word2vec_format` function to load it into memory. You can easily download the binary from an Amazon AWS repository using the `wget` command from a shell:

```
wget -c "https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz"
```

After downloading and decompressing the file, you can use it with the Gensim `KeyedVectors` functions:

```
import gensim
model = gensim.models.KeyedVectors.load_word2vec_format(
'GoogleNews-vectors-negative300.bin.gz', binary=True)
```

Now, we can easily get the vector of a word by calling model[word]. However, a problem arises when we are dealing with sentences instead of individual words. In our case, we need vectors for all of question1 and question2 in order to come up with some kind of comparison. For this, we can use the following code snippet. The snippet basically adds the vectors for all words in a sentence that are available in the Google news vectors and gives a normalized vector at the end. We can call this sentence to vector, or Sent2Vec.

Make sure that you have **Natural Language Tool Kit** (**NLTK**) installed before running the preceding function:

```
$ pip install nltk
```

It is also suggested that you download the `punkt` and `stopwords` packages, as they are part of NLTK:

```
import nltk
nltk.download('punkt')
nltk.download('stopwords')
```

If NLTK is now available, you just have to run the following snippet and define the `sent2vec` function:

```
from nltk.corpus import stopwords
from nltk import word_tokenize
stop_words = set(stopwords.words('english'))

def sent2vec(s, model):
M = []
words = word_tokenize(str(s).lower())
for word in words:
```

```
        #It shouldn't be a stopword
        if word not in stop_words:
        #nor contain numbers
        if word.isalpha():
        #and be part of word2vec
        if word in model:
        M.append(model[word])
        M = np.array(M)
        if len(M) > 0:
        v = M.sum(axis=0)
        return v / np.sqrt((v ** 2).sum())
        else:
        return np.zeros(300)
```

When the phrase is null, we arbitrarily decide to give back a standard vector of zero values.

To calculate the similarity between the questions, another feature that we created was word mover's distance. Word mover's distance uses Word2vec embeddings and works on a principle similar to that of earth mover's distance to give a distance between two text documents. Simply put, word mover's distance provides the minimum distance needed to move all the words from one document to another document.

The WMD has been introduced by this paper: *"KUSNER, Matt, et al. From word embeddings to document distances. In: International Conference on Machine Learning. 2015. p. 957-966"* which can be found at http://proceedings.mlr.press/v37/kusnerb15.pdf. For a hands-on tutorial on the distance, you can also refer to this tutorial based on the Gensim implementation of the distance: https://markroxor.github.io/gensim/static/notebooks/WMD_tutorial.html
Final **Word2vec** (**w2v**) features also include other distances, more usual ones such as the Euclidean or cosine distance. We complete the sequence of features with some measurement of the distribution of the two document vectors:

1. Word mover distance
2. Normalized word mover distance
3. Cosine distance between vectors of question1 and question2
4. Manhattan distance between vectors of question1 and question2
5. Jaccard similarity between vectors of question1 and question2
6. Canberra distance between vectors of question1 and question2
7. Euclidean distance between vectors of question1 and question2
8. Minkowski distance between vectors of question1 and question2
9. Braycurtis distance between vectors of question1 and question2
10. The skew of the vector for question1
11. The skew of the vector for question2
12. The kurtosis of the vector for question1
13. The kurtosis of the vector for question2

All the Word2vec features are denoted by **fs4**.

A separate set of w2v features consists in the matrices of Word2vec vectors themselves:

1. Word2vec vector for question1
2. Word2vec vector for question2

These will be represented by **fs5**:

```
w2v_q1 = np.array([sent2vec(q, model)
                   for q in data.question1])
w2v_q2 = np.array([sent2vec(q, model)
                   for q in data.question2])
```

In order to easily implement all the different distance measures between the vectors of the Word2vec embeddings of the Quora questions, we use the implementations found in the `scipy.spatial.distance module`:

```
from scipy.spatial.distance import cosine, cityblock,
          jaccard, canberra, euclidean, minkowski, braycurtis
```

```
      data['cosine_distance'] = [cosine(x,y)
      for (x,y) in zip(w2v_q1, w2v_q2)]
      data['cityblock_distance'] = [cityblock(x,y)
      for (x,y) in zip(w2v_q1, w2v_q2)]
      data['jaccard_distance'] = [jaccard(x,y)
      for (x,y) in zip(w2v_q1, w2v_q2)]
      data['canberra_distance'] = [canberra(x,y)
      for (x,y) in zip(w2v_q1, w2v_q2)]
      data['euclidean_distance'] = [euclidean(x,y)
      for (x,y) in zip(w2v_q1, w2v_q2)]
      data['minkowski_distance'] = [minkowski(x,y,3)
      for (x,y) in zip(w2v_q1, w2v_q2)]
      data['braycurtis_distance'] = [braycurtis(x,y)
      for (x,y) in zip(w2v_q1, w2v_q2)]
```

All the features names related to distances are gathered under the list `fs4_1`:

```
fs4_1 = ['cosine_distance', 'cityblock_distance',
         'jaccard_distance', 'canberra_distance',
         'euclidean_distance', 'minkowski_distance',
         'braycurtis_distance']
```

The Word2vec matrices for the two questions are instead horizontally stacked and stored away in the `w2v` variable for later usage:

```
w2v = np.hstack((w2v_q1, w2v_q2))
```

The Word Mover's Distance is implemented using a function that returns the distance between two questions, after having transformed them into lowercase and after removing any stopwords. Moreover, we also calculate a normalized version of the distance, after transforming all the Word2vec vectors into L2-normalized vectors (each vector is transformed to the unit norm, that is, if we squared each element in the vector and summed all of them, the result would be equal to one) using the `init_sims` method:

```
def wmd(s1, s2, model):
    s1 = str(s1).lower().split()
    s2 = str(s2).lower().split()
    stop_words = stopwords.words('english')
    s1 = [w for w in s1 if w not in stop_words]
    s2 = [w for w in s2 if w not in stop_words]
    return model.wmdistance(s1, s2)
```

```
    data['wmd'] = data.apply(lambda x: wmd(x['question1'],
    x['question2'], model), axis=1)
    model.init_sims(replace=True)
    data['norm_wmd'] = data.apply(lambda x: wmd(x['question1'],
    x['question2'], model), axis=1)
    fs4_2 = ['wmd', 'norm_wmd']
```

After these last computations, we now have most of the important features that are needed to create some basic machine learning models, which will serve as a benchmark for our deep learning models. The following table displays a snapshot of the available features:

question1	What is the story of Kohinoor (Koh-i-Noor) Dia...
question2	What would happen if the Indian government sto...
is_duplicate	0
len_q1	51
len_q2	88
diff_len	-37
len_char_q1	21
len_char_q2	29
len_word_q1	8
len_word_q2	13
common_words	4
fuzz_qratio	66
fuzz_WRatio	86
fuzz_partial_ratio	73
fuzz_partial_token_set_ratio	100
fuzz_partial_token_sort_ratio	75
fuzz_token_set_ratio	86
fuzz_token_sort_ratio	63
wmd	3.77235
norm_wmd	1.3688
cosine_distance	0.512164
cityblock_distance	14.1951
jaccard_distance	1
canberra_distance	177.588
euclidean_distance	1.01209
minkowski_distance	0.45591
braycurtis_distance	0.592655
skew_q1vec	0.00873466
skew_q2vec	0.0947038
kur_q1vec	0.28401
kur_q2vec	-0.034444

Let's train some machine learning models on these and other Word2vec based features.

Testing machine learning models

Before proceeding, depending on your system, you may need to clean up the memory a bit and free space for machine learning models from previously used data structures. This is done using `gc.collect`, after deleting any past variables not required anymore, and then checking the available memory by exact reporting from the `psutil.virtualmemory` function:

```
import gc
import psutil
del([tfv_q1, tfv_q2, tfv, q1q2,
     question1_vectors, question2_vectors, svd_q1,
     svd_q2, q1_tfidf, q2_tfidf])
del([w2v_q1, w2v_q2])
del([model])
gc.collect()
psutil.virtual_memory()
```

At this point, we simply recap the different features created up to now, and their meaning in terms of generated features:

- `fs_1`: List of basic features
- `fs_2`: List of fuzzy features
- `fs3_1`: Sparse data matrix of TFIDF for separated questions
- `fs3_2`: Sparse data matrix of TFIDF for combined questions
- `fs3_3`: Sparse data matrix of SVD
- `fs3_4`: List of SVD statistics
- `fs4_1`: List of w2vec distances
- `fs4_2`: List of wmd distances
- `w2v`: A matrix of transformed phrase's Word2vec vectors by means of the `Sent2Vec` function

We evaluate two basic and very popular models in machine learning, namely logistic regression and gradient boosting using the `xgboost` package in Python. The following table provides the performance of the logistic regression and `xgboost` algorithms on different sets of features created earlier, as obtained during the Kaggle competition:

Feature set	**Logistic regression accuracy**	**xgboost accuracy**
Basic features (fs1)	0.658	0.721
Basic features + fuzzy features (fs1 + fs2)	0.660	0.738
Basic features + fuzzy features + w2v features (fs1 + fs2 + fs4)	0.676	0.766
W2v vector features (fs5)	*	0.78
Basic features + fuzzy features + w2v features + w2v vector features (fs1 + fs2 + fs4 + fs5)	*	**0.814**
TFIDF-SVD features (`fs3-1`)	0.777	0.749
TFIDF-SVD features (`fs3-2`)	0.804	0.748
TFIDF-SVD features (`fs3-3`)	0.706	0.763
TFIDF-SVD features (`fs3-4`)	0.700	0.753
TFIDF-SVD features (`fs3-5`)	0.714	0.759

* = These models were not trained due to high memory requirements.

We can treat the performances achieved as benchmarks or baseline numbers before starting with deep learning models, but we won't limit ourselves to that and we will be trying to replicate some of them.

We are going to start by importing all the necessary packages. As for as the logistic regression, we will be using the scikit-learn implementation.

The xgboost is a scalable, portable, and distributed gradient boosting library (a tree ensemble machine learning algorithm). Initially created by Tianqi Chen from Washington University, it has been enriched with a Python wrapper by Bing Xu, and an R interface by Tong He (you can read the story behind xgboost directly from its principal creator at homes.cs.washington.edu/~tqchen/2016/03/10/story-and-lessons-behind-the-evolution-of-xgboost.html). The xgboost is available for Python, R, Java, Scala, Julia, and C++, and it can work both on a single machine (leveraging multithreading) and in Hadoop and Spark clusters.

Detailed instruction for installing xgboost on your system can be found on this page: github.com/dmlc/xgboost/blob/master/doc/build.md

The installation of xgboost on both Linux and macOS is quite straightforward, whereas it is a little bit trickier for Windows users.

For this reason, we provide specific installation steps for having xgboost working on Windows:

1. First, download and install Git for Windows (git-for-windows.github.io)
2. Then, you need a MINGW compiler present on your system. You can download it from www.mingw.org according to the characteristics of your system
3. From the command line, execute:

   ```
   $> git clone --recursive https://github.com/dmlc/xgboost
   $> cd xgboost
   $> git submodule init
   $> git submodule update
   ```
4. Then, always from the command line, you copy the configuration for 64-byte systems to be the default one:

   ```
   $> copy makemingw64.mk config.mk
   ```

   Alternatively, you just copy the plain 32-byte version:

   ```
   $> copy makemingw.mk config.mk
   ```
5. After copying the configuration file, you can run the compiler, setting it to use four threads in order to speed up the compiling process:

   ```
   $> mingw32-make -j4
   ```
6. In MinGW, the `make` command comes with the name `mingw32-make`; if you are using a different compiler, the previous command may not work, but you can simply try:

   ```
   $> make -j4
   ```
7. Finally, if the compiler completed its work without errors, you can install the package in Python with:

   ```
   $> cd python-package
   $> python setup.py install
   ```

If xgboost has also been properly installed on your system, you can proceed with importing both machine learning algorithms:

```
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
import xgboost as xgb
```

Since we will be using a logistic regression solver that is sensitive to the scale of the features (it is the `sag` solver from https://github.com/EpistasisLab/tpot/issues/292, which requires a linear computational time in respect to the size of the data), we will start by standardizing the data using the `scaler` function in scikit-learn:

```
scaler = StandardScaler()
y = data.is_duplicate.values
y = y.astype('float32').reshape(-1, 1)
X = data[fs_1+fs_2+fs3_4+fs4_1+fs4_2]
X = X.replace([np.inf, -np.inf], np.nan).fillna(0).values
X = scaler.fit_transform(X)
X = np.hstack((X, fs3_3))
```

We also select the data for the training by first filtering the `fs_1`, `fs_2`, `fs3_4`, `fs4_1`, and `fs4_2` set of variables, and then stacking the `fs3_3` sparse SVD data matrix. We also provide a random split, separating 1/10 of the data for validation purposes (in order to effectively assess the quality of the created model):

```
np.random.seed(42)
n_all, _ = y.shape
idx = np.arange(n_all)
np.random.shuffle(idx)
n_split = n_all // 10
idx_val = idx[:n_split]
idx_train = idx[n_split:]
x_train = X[idx_train]
y_train = np.ravel(y[idx_train])
x_val = X[idx_val]
y_val = np.ravel(y[idx_val])
```

As a first model, we try logistic regression, setting the regularization l2 parameter C to 0.1 (modest regularization). Once the model is ready, we test its efficacy on the validation set (`x_val` for the training matrix, `y_val` for the correct answers). The results are assessed on accuracy, that is the proportion of exact guesses on the validation set:

```
logres = linear_model.LogisticRegression(C=0.1,
                                 solver='sag', max_iter=1000)
logres.fit(x_train, y_train)
lr_preds = logres.predict(x_val)
log_res_accuracy = np.sum(lr_preds == y_val) / len(y_val)
print("Logistic regr accuracy: %0.3f" % log_res_accuracy)
```

After a while (the solver has a maximum of 1,000 iterations before giving up converging the results), the resulting accuracy on the validation set will be 0.743, which will be our starting baseline.

Now, we try to predict using the `xgboost` algorithm. Being a gradient boosting algorithm, this learning algorithm has more variance (ability to fit complex predictive functions, but also to overfit) than a simple logistic regression afflicted by greater bias (in the end, it is a summation of coefficients) and so we expect much better results. We fix the max depth of its decision trees to 4 (a shallow number, which should prevent overfitting) and we use an eta of 0.02 (it will need to grow many trees because the learning is a bit slow). We also set up a watchlist, keeping an eye on the validation set for an early stop if the expected error on the validation doesn't decrease for over 50 steps.

It is not best practice to stop early on the same set (the validation set in our case) we use for reporting the final results. In a real-world setting, ideally, we should set up a validation set for tuning operations, such as early stopping, and a test set for reporting the expected results when generalizing to new data.
After setting all this, we run the algorithm. This time, we will have to wait for longer than we when running the logistic regression:

```
params = dict()
params['objective'] = 'binary:logistic'
params['eval_metric'] = ['logloss', 'error']
params['eta'] = 0.02
params['max_depth'] = 4
d_train = xgb.DMatrix(x_train, label=y_train)
d_valid = xgb.DMatrix(x_val, label=y_val)
watchlist = [(d_train, 'train'), (d_valid, 'valid')]
bst = xgb.train(params, d_train, 5000, watchlist,
                early_stopping_rounds=50, verbose_eval=100)
xgb_preds = (bst.predict(d_valid) >= 0.5).astype(int)
xgb_accuracy = np.sum(xgb_preds == y_val) / len(y_val)
print("Xgb accuracy: %0.3f" % xgb_accuracy)
```

The final result reported by `xgboost` is `0.803` accuracy on the validation set.

Building TensorFlow model

The deep learning models in this article are built using TensorFlow, based on the original script written by Abhishek Thakur using Keras (you can read the original code at https://github.com/abhishekkrthakur/is_that_a_duplicate_quora_question). Keras is a Python library that provides an easy interface to TensorFlow. Tensorflow has official support for Keras, and the models trained using Keras can easily be converted to TensorFlow models. Keras enables the very fast prototyping and testing of deep learning models. In our project, we rewrote the solution entirely in TensorFlow from scratch anyway.

To start, let's import the necessary libraries, in particular, TensorFlow, and let's check its version by printing it:

```
import zipfile
from tqdm import tqdm_notebook as tqdm
import tensorflow as tf
print("TensorFlow version %s" % tf.__version__)
```

At this point, we simply load the data into the `df` pandas dataframe or we load it from disk. We replace the missing values with an empty string and we set the `y` variable containing the target answer encoded as 1 (duplicated) or 0 (not duplicated):

```
try:
    df = data[['question1', 'question2', 'is_duplicate']]
except:
    df = pd.read_csv('data/quora_duplicate_questions.tsv',
                                                sep='t')
```

```
    df = df.drop(['id', 'qid1', 'qid2'], axis=1)

df = df.fillna('')
y = df.is_duplicate.values
y = y.astype('float32').reshape(-1, 1)
```

To summarize, we built a model with the help of TensorFlow in order to detect duplicated questions from the Quora dataset.

To know more about how to build and train your own deep learning models with TensorFlow confidently, do checkout *this book TensorFlow Deep Learning Projects.*

Read Next:

- TensorFlow 1.9.0-rc0 release announced
- Implementing feedforward networks with TensorFlow
- How TFLearn makes building TensorFlow models easier

Sunith Shetty

Data Science fanatic. Cricket fan. Series Binge watcher. You can find me hooked to my PC updating myself constantly if I am not cracking lame jokes with my team.