# NLP Final Paper: Question pair duplicate

Quang Nguyen

December 2018

## 1 Introduction

In this final project, I will tackle one of the most popular problem in Natural Language Processing - analyzing sentence semantic. To be more specific, I will take a look at how some of the most popular methods for analyzing sentence semantic work in identifying duplicate questions. This is a very useful problem for question based company like Quora, Stack Overflow and even Google. The ability to identify duplicate question will help those company to group all questions with the same meaning in one thread, avoid having multiple threads spread out for similar questions. Although this problem can be solve in numerous ways, this paper will take a look at how I extract the semantic meaning from the question, then using one of the machine learning classifier model, I will be able to train question pairs and then predict the result for new pair of questions.

The question can be defined formally as:

Given a dataset $D$ that contain $q1_id$, $q2_id$, $question1$, $question2$, $isduplicate$ as columns name, we need to predict the $isduplicate$ column in a new dataset $T$.

Thus, I need to create a classifier model $f$ such that given a $q1$ and $q2$ return the probability that these two questions are similar.

## 2 Related Work

There have been a lot of works that aim to solve this problem. Some of the approach that I have read are: Cosine Similarity, Jaccard, k-Shingling, simple machine learning model, word embedding, and the most advance method is using recurrent neutral network to classify duplicate question. First, I take a look at how Cosine Similarity works. Suppose I can represent each question as a vector, thus I represent question1 as the vector $q1$, question2 as the vector $q2$. Thus, the cosine similarity between the two vectors is $cos(q1, q2) = \dfrac{q1 \cdot q2}{|q1||q2|}$. Therefore, the cosine between two questions can be determined by the dot product between the two vectors and the product of its size. This is really convenience since the cosine represents how small the angle between the two vector. The intuition is, the smaller the angle between two vectors, the closer they are with each other. Thus, if the angle is 90 degree, then the two vectors are completely different, however, if the angle is 0 degree then the two vectors are the same. Thus, cosine can be used to determine if the two question vectors are similar to each other, if the cosine of the two questions are close to 1 that means they are likely to be similar, otherwise if the cosine similarity are close to 0, then I have a very two different questions. The cosine similarity is a simple approach and it is very easy to implement, however, using cosine similarity alone cannot help me in understanding the semantic of the question. Jaccard is another interesting approach. Jaccard looks for the intersection between the two question - how many words appear in both question 1 and question2 over how many different words appear in both question. Thus, Jaccard similarity can be defined as a $J(Q_1, Q_2) = \frac{Q_1 \cap Q_2}{Q_1 \cup Q_2}$. Although Jaccard similarity is a really good technique to identify questions with similar words share, it does not show the semantics between the two questions. It is really likely that questions that are totally different from each other may share a lot of similar words. For example, if $q1$ is "How to properly feed your cat" and $q2$ is "How to properly feed your dogs", the Jaccard similarity, although really high, will not have the real understanding of the questions. Last but not least, the k-Shingling method is another powerful tool in determine similar documents. The k-Shingling is some what similar to the Jaccard similarity. Instead of looking at the whole document and determine the number of shared word, the k-Shingling looks at the window of $n$ consecutive sequence within a document. Next, it will try to find if that window of sequence appears any where else in another document. The k-Shingling has the same problem with other previous model, it fails to analyze the semantic of the document. My method will go away from these naive approach and use word embedding. Word embedding is a method that transforms words to vectors. The method will be able to show a clear distinction between words with different meaning (for example: cat and dog) and synonym words. My approach will also use a machine learning classifier to predict probability that two questions are duplicated. Also, Jaccard similarity and cosine similarity will be added to the model to improve its performance.

## 3 Method

Here I will show the detail of the words embedding algorithm. There are two different word embedding technique that I will use in my solution, the first one is the tf-idf algorithm and the other one is the word2Vec embedding.

**TF-IDF**[1]: The tf-idf algorithm calculates the weighted value of a word $i$ given the document $j$. The weighted value can be formally represented as $w_{ij} = tf_{ij}.idf_i$. Let's analyze this equation to have a better understanding of the weighted value. $tf_{ij}$, otherwise known as the term frequency of the word, denotes the frequency of word $i$ in the document $j$. Therefore, the larger the number of time word $i$ appear in the document $j$, the higher the $tf_{ij}$ value, the higher the weighted value $w_{ij}$. However, there is a problem with the $tf_{ij}$ since punctuation and stop word like *the*, *and*, *a* can appear a lot in the document. Thus those word will have a really high weighted value. However, stop words and punctuation are not very valuable information for the document. I do not want word like this to have a higher weight. The variable $idf_i$ calculates the inverse document frequency of the word. The inverse frequency of a word measures how likely the word appear in the document $_{,i} = \frac{N}{df_i}$ where $N$ is the total number of documents and $df_i$ is the total number of document contain word $i$. Thus, the more popular word like *the*, *and* will have a lower weighted value. Using this weighed value, I will calculate the tf-idf between each word in the document. Thus in the end I will have a set of vector with a size of $|V|$, where $|V|$ is the vocab size. Each vector represents the word and each cell in the vector represents the weighted tf-idf value of that word with another word.

One of the problem of the TF-IDF algorithm is that it creates a word vector with very high dimension and lots of zeros. This is because usually, a document will have a lot of different words, thus the vector has high dimension and not any two words in the document will be near to each other, thus, lots of cell in the vector would be zeros. As a result this kind of approach really hurt my computational time and are unlikely to be used in a real life problem. One of the more popular approach for word embedding is the word2Vec embedding technique.

**Word2Vec[1]** Word2Vec creates a word with a lower dimension because it assumes before hand that any word should be starting with a vectors of size $D$, where $D$ can be hand picked by human. The intuition of this embedding technique is that it will look at the document, slightly change the vector of each word so that it is similar to some of the word next to it. The word2Vec algorithm will keep changing the vector of the word until no further improvement can be done.

The details of the word2Vec can be represented as follow:

1. Determine a window size of $n$, a window size means for each of the word in the document. We will take a look at $n$ words before this word, which we will call a target word and $n$ words after the target word as the context words. We will call this set of target word, context word $C+$

2. Next we will select randomly from the documents other words outside the window range to select words that are not similar to the target word. We will call this set of target word, context word $C-$

3. Thus $P(+|t,c)$, the probability that word $t$ and word $c$ are similar to each other, is represented as $sigmoid(t \cdot c)$. Similarly, the probability that word $t$ and $c$ are not similar to each other is $P(-|t,c)$

4. Finally we have a lost function:
   $L(\theta) = \sum_{(t,c)\in C+} logP(+|t,c) + \sum_{(t,c)\in C-} logP(-|t,c)$

We want to minimize this function. Thus, we use a gradient descent algorithm so that each and every time, the gradient descent algorithm modifies the word vector so that the lost function $L(\theta)$ achieve its local maxima.

5. the algorithm return a matrix $T$ of target word, where each entry is a $1 \times D$ vector $t_i$. Since we are allow to pick $D$ at the beginning of the word2Vec algorithm, we can pick $D$ reasonably small.

The word2Vec embedding technique returns a vector for every word in the document. The result shows that word2Vec returns a really dense vector, which can be beneficial for the sentiment analysis.

The word2Vec and the tf-idf algorithm only shows you the vector of each and every words in the question. It does not show you how to combine word vectors to represent the vector of a sentence. One of the most simple solution is to use the centroid technique, which is summing up all the words vectors and divide by its total number of word. There for for each question, I will convert each word from the question to vector, summing up all the vectors inside that question then divided by the total number of word for a question.

The next step is to pick a classifier. For this project, I use two different classifier, the Logistic Regression and the Naive Bayes Regression.

For Logistic Regression, it uses gradient descent to train logistic regression log loss function $J(\theta)$. For my Logistic Regression model, I pick the independence variable to be the absolute value of the different between two vectors. Thus if question1 is represented as $q1$ vector, question2 is represented as $q2$ vector, then my independence variable $X$ is $abs(q1 - q2)$. Our hope is that if the two question are similar to each other, then my independence variable is close to the zero vector.

For Naive Bayes classifier, it uses a probabilistic function to determine if this a document is in a particular class. For this problem, I determine the document as the concatenation of the two questions. I have two classes for the Naive Bayes classifier, the first class indicates that the concatenate of the two questions is in a class of duplicated, which means the two question are the same. The second class indicates that that concatenate of the two questions is in a class of non duplicated, which means that the two questions are not the same.

## 4 Evaluation

Both the Naive Bayes and the Logistics Regression achieve roughly $68\% - 70\%$ for accuracy. However, both of the model give e a really low percentage in precision, only 10% for the Naive Bayes and 20% for the Logistic Regression. When I take a look at all the data that I predict wrong, I found out that a large percentage of the wrong prediction is because the model fails to identify similarity question. Both model works well with questions that are different from each other. There are two approaches that I could take in order to improve the performance of the model.

Firstly, I increase the amount of data that I am using for the current model. Currently, I am using around 4000 question pair for the training process and another 1000 questions pair for the testing process. I will use a much bigger dataset which contains $300,000$ questions pair for the training and $100,000$. Despite my effort in increasing the number of data for my model, the accuracy, precision and recall do not seems to improve. It is suggest that the previous statistics is the best result that I can have for this current model. Which is why this lead to my second approach in improving the model. I have to completely change the independence variable of each model in order to improve performance. Some of the new independence variable that I have are :word_match [2], weight_tf-idf [2] and word2vec_dot_prod. word_match is essentially the Jaccard similarity value that I have talked earlier. It is the counting of the words that appear in both question, divided by the total number of words in the two questions. weight_tf-idf is the weighted tf-idf value of each of the shared word in the question, divided by the total sum of weighted tf-idf value of all the word in the question. Lastly, word2vec_dot_prod is the dot product between $q1$ and $q2$, where $q1$ is the vector representation of question1, $q2$ is the vector representation of question2.

My last approach in improving the model will be using some of the most powerful machine learning library in machine learning application. The XGB library stands for eXtreme Gradient Boosting algorithm [2]. This library focus on maximizing the computational speed and model performance of a machine learning model. For intuition, the XGB algorithm adds new models to the existing old model during the training process, in order to fix the residual errors made by the previous model. Suppose our model start with a really simple model $F_m$, at each of the iteration of the Gradient descent boosting, the model at a new function $h$ so that the new model is $F_{m+1} = F_m(x)+h(x)$. The gradient descent boosting assume that the new model will be close to the dependence variable $y$. Thus, we want a function $h$ such that $F_m(x) + h(x) = F_{m+1} = y$. Thus, $h(x) = y - F_m$ and we will fit $h$ to the residual error $y - F_m(x)$. As a result, each new model $F_{m+1}$ attempt to fix the residual error of the previous model.

Finally, with the help of the XGB algorithm, I use it for my Logistic Regression model and the three new variables. In the end, I achieve a total of 73% accuracy, 43% precision and 53% for recall. It is a huge improvement over the accuracy and the precision value. Although I have to sacrifice a little percentage for recall, the improvement of the precision value is totally worth the deduction.

## 5 Discussion

My final result with be the one using Gradient boosting algorithm on the Logistic Regression, my independence variable will be the value of word_match, weight_tf-idf and word2vec_dot_prod between the two question. I train this data over $300,000$ question pair and test it on over $100,000$ question pair. My result shows that the model is really good at identifying questions that are not duplicated from each other. One of the most impressing result that I have is question pair where the structure of the question is identical with each other, but one of the key noun or noun phrase is totally different (eg: How can I feed my dog? vs How can I feed my cat?). For those question pair, the model correctly identify that those pair are not duplicated. My model do not work really good on question that is similar to each other but is paraphrase totally different. I will have to have a better sentence semantic in order to tackle those problems. One of the technique that other people use is the Long Short Term Memory (LSTM) technique. LSTM is a recurrent neutral network that helps us to remember some of the context within the question, but also forgot a chunk of information that is far from the context.

## 6 References

[1] Dan Jurafsky and James H. Martin. Speech and Language Processing. Sep 23, 2018.
[2] Mikel Bober-irizar, Data Analysis  XGBoost Starter (0.35460 LB),
https://www.kaggle.com/anokas/data-analysis-xgboost-starter-0-35460-lb