Building a Text Matching System for Question Matching

by Ankita Choudhury

Abstract

Building a Text matching system has a high practical significance for retrieving relevant text from a large number of documents. Three different methods namely TFIDF, word average embedding method and inverse document frequency method were used to build a text matching system. The systems were tested on the first 100 questions which were duplicate. A maximum accuracy score of 77% and 67% in top5 and top 2 matches was obtained using average word model.

Introduction

Text matching is a part of Natural Language Processing (NLP). NLP helps computers read, understand and interpret human language. It is being widely used for sentiment analysis, language translation, text extraction, chatbots etc. With the increasing number of online platforms containing comments, reviews, question and answer forums etc., the task of matching different texts has high practical significance. It can be used to find a similar document or to find the answer to a question. Three different types of algorithms are used to perform text matching. TF-IDF is a simple yet efficient method to perform text matching. It uses a term document matrix. The other two methods, word average and smooth inverse frequency use pretrained word embeddings to create sentence embedding.

Dataset

The dataset used for this project is a collection of questions. The data is a tsv file named "data.csv" which contains six columns namely, Id, qid1, qid2, question1, question2 and is_duplicate. The column is_duplicate indicates if question 1 and question are duplicate or same. It has two values, 0 and 1. There are 363181 and 363180 questions in question 1 and question 2 respectively. After removing duplicated questions, the class distribution of label observed is shown in Figure 1. There are 63108 reviews labelled as duplicate, 169095 as not duplicate. The classes 0 and 1 are not balanced in the dataset with more than half samples labelled as 1. For training, the questions in question 2 are used. The first 100 questions where is_duplicate is equal to 1 are used for testing. Text matching scores are calculated for each question 1 in the testing test against all questions in question 2. Different evaluation metrics are used for different techniques which are discussed later on in this report.

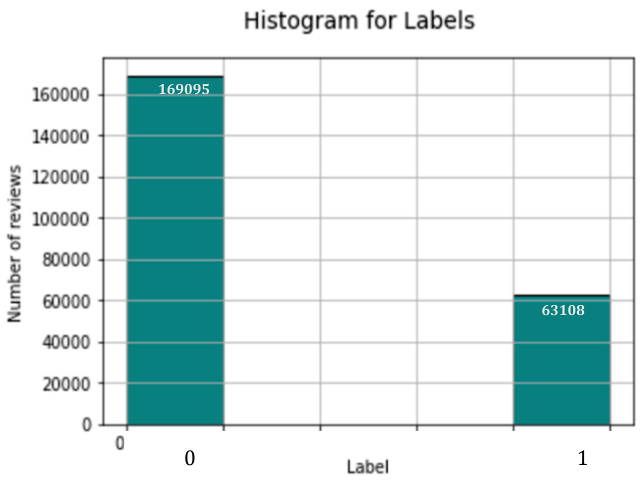


Figure 1: Distribution of classes

Data Pre-Processing

A Jupyter notebook with Python kernel was used to perform all the tests on Google Collab. Natural Language Toolkit (NLTK) was the primary library used for processing text. NLTK is one of the most popular libraries used for Natural Language Processing. NLTK was used to perform tokenization, lemmatization, stemming, stop word removal etc. Sklearn, Pandas and NumPy library were also frequently used for data loading, manipulation and transformation. Matplotlib library was used to plot visualisations. The csv file was loaded and converting to a data frame using read_csv function of pandas. The data type of 'id' and 'qid1' were changed from object to numeric type. A few null values were found in the dataset which were dropped. There were 97251 and 90208 duplicated questions in column 'question1' and 'question2' respectively. Rows which contained these duplicated values were removed. A test data frame was defined which contains all the first 100 rows with "is_duplicate" equal to 1.

Tokenisation and Stop word removal

A function to perform text cleaning was implemented. The function ‘clean_corpus’ first converts all input text to lower case. This is done as models are case sensitive. Converting all text to lower case might be counterproductive for some words like ‘US’ as they lose semantic their meaning. Overall, it is still a good method for pre-processing text. Then the text data was tokenized using word_tokenize function from NLTK. Stop words are the commonly used words in the English language. They carry little meaning and have low importance for predictions. The stop words were removed from the tokenized text. The stop words that were removed were predefined stop words in the English language by NLTK library. After that, all the punctuations were also removed from the tokens. The text which was numerical was also removed.

Stemming

Stemming was performed next. Stemming is a text normalisation technique. Stemming reduces the words to their stem or root form [1]. The root form is achieved by removing affixes like -ize, -ed, -de etc. The stem obtained after stemming a word might not be a valid word in the same language. Another problem with stemming is that different meaning words might be reduced to the same stem for example ‘universe’ and ‘university’ to ‘univers’. Different words which have the same stem might get reduced to different stems. Stemming was used to pre-process text for TF-IDF implementation. It was useful to reduce the vocabulary size for training and hence train faster.

Lemmatization

Lemmatization is the process of finding the lemma of a word based on its meaning. Lemmatization aims to remove inflectional endings. It aids in returning a word's base form. NLTK provides a lemmatizer based on WordNet built in morphy function. If any word is not found in Wordnet, it returns the word unchanged. The function ‘clean_corpus_2’ performs lemmatization instead of stemming. Lemmatization with no stemming was used to preprocess text for creating sentence embeddings. As sentence embedding methods use a pretained word embedding model to vectorize a sentence, lemmatization was a good method to preprocess text as it always produces existing words in the English language.

Removing Words with very high word count

All the processed tokens were joined together. The word count was calculated for the whole dataset and the words with very high frequency precisely, words that appeared more than 5000 times in the data were removed for TFIDF method. The words which appear many times in the whole corpus have low importance for a particular document. After all this pre-processing, if some questions were only empty strings, then those questions were also removed.

TF-IDF and Inverted Index

TF-IDF stands for Term Frequency - Inverse Document Frequency. TF IDF is used to quantify words in a document by assigning a weight to each word which represents the importance of the word in the document. The formula for calculating TF IDF is

$$\text{TF-IDF} = \text{Term Frequency (TF)} * \text{Inverse Document Frequency (IDF)}$$

Term Frequency (TF)

This metric counts the number of times a word appears in a document. Certain words like ‘the’, ‘and’, ‘is’ etc. appear many times in a document. Thus, they have a high frequency.

$$\text{Term Frequency of a document} = \text{Count} / \text{Total number of words}$$

To calculate term frequency for a document, the frequency of a word in a document is divided by the total number of words in the document. This is a way to normalize the frequency for a document. Documents containing many words would not have greater importance than a document containing a smaller number of words.

Inverse Document Frequency (IDF)

A document frequency of a word is the number of occurrences of it in different documents. It is the number of documents in which a word appears. The number of times a word appears in a single document is not important. If a word appears at least once in a document, the document frequency is 1. If the word is present at least once in 2 documents, the document frequency is 2 and so on.

Inverse Document Frequency = Log (Count of Documents / Document Frequency)

Inverse Document Frequency of a word is calculated by dividing the total number of documents by the document frequency of a word. Since common words like stop words appear a lot of times in a large number of documents, their IDF would still be small number. Usually, the logarithm of the IDF value is used. This is done because the IDF value computed might be too small and might be interpreted as zero by the computer.

Inverted Index

An inverted index is used to map a word to its location. It is useful as it has a very fast computational speed for searching a word compared to other methods. A dictionary is used to implement inverted index. Inverted index stores the name of the word as key and the number of document/documents it appears in as the value.

Implementation

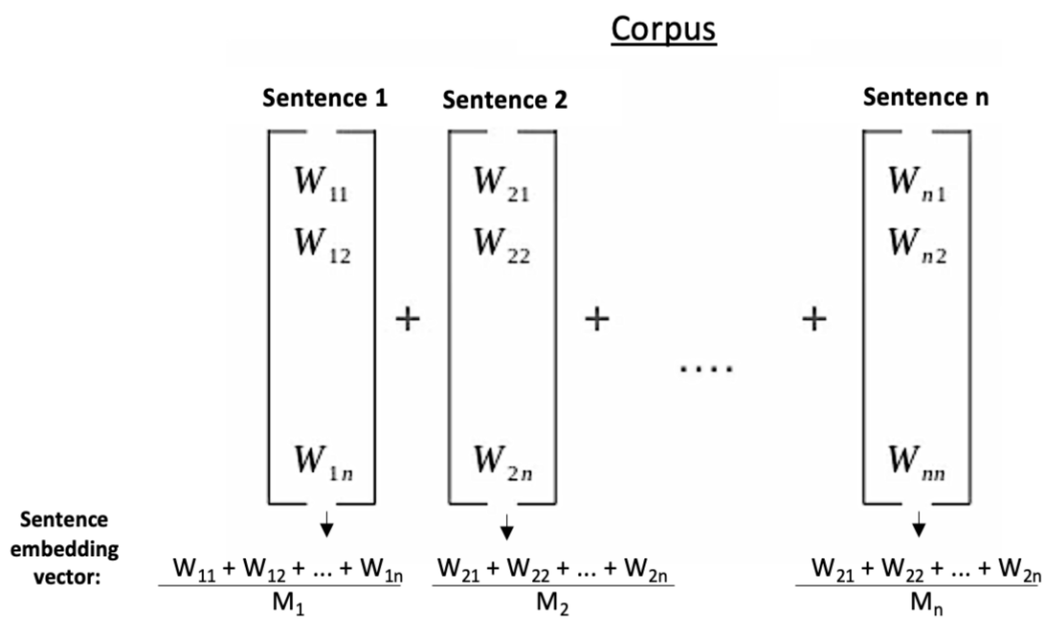
To compute the term frequency, first a vocabulary is formed. The vocabulary is a dictionary which contains the word as key and its frequency as value. It is formed using all the words from all documents in the training data i.e., ‘question2’ column values. Then, for each question in ‘question2’, the frequency of a word in a question is calculated. Words with length less than 2 are ignored. If a word in vocabulary exists, store the question the word belongs. Then the term frequency is calculated for every row of ‘question2’. After computing term frequency, Term Frequency (TF) and Inverse Document Frequency (IDF) are multiplied to get the TFIDF value.

A TFIDF matrix is defined which contains the frequency of word as column values and the question a word belongs to as row values. Each row represents a question, and each column represents word. An inverted index is implemented next. Inverted index dictionary stored the name of the word as key and the number of documents the word appeared in as the value.

After defining TF-IDF and inverted index, a function is defined to calculate similarity between input question and the first 50k questions in ‘question2’. The input text is first pre-processed then split into words. If a word if present in inverted index keys, then the value for that key are the document numbers that the word appears in. If the word is present in the vocabulary, then the TF-IDF value from the TFIDF matrix is the score for that word. Similarly, TF-IDF values for all words in a single question are added up to give the question's final score. The questions are ranked in descending order of their score, thus the question with the higher score is more similar to the input question.

Sentence Embedding by averaging Word Embedding

Averaging a pre- trained word embedding vectors to create sentence embedding vectors is another way to perform text matching. The pre trained embedding used for this project is the Glove 100-dimensional word embedding. This embedding contained 400000 words and their vector representation. Glove is an unsupervised learning algorithm which can maps words together in a space using semantic similarity. It is open sourced and can be downloaded from “http://nlp.stanford.edu/data/glove.6B.zip”. The process of averaging word embeddings is described below



Corpus
Sentence 1
Sentence 2
Sentence n
W_{11}
W_{12}
W_{1n}
W_{21}
W_{22}
W_{2n}
W_{n1}
W_{n2}
W_{nn}
+
+
....
+
Sentence embedding vector:
$\frac{W_{11} + W_{12} + \ldots + W_{1n}}{M_1}$
$\frac{W_{21} + W_{22} + \ldots + W_{2n}}{M_2}$
$\frac{W_{21} + W_{22} + \ldots + W_{2n}}{M_n}$

Figure 2: Sentence embedding by averaging word embedding formula

Here M_1, M_2, … , M_n are the total number of words in the sentence 1, sentence 2 , …, sentence n. Every sentence is tokenised into words. The pretrained vector embedding for each word in a sentence is stored. W_{11}, W_{12}, … W_{1n} represent all the word vectors in sentence 1. These vectors have a consistent dimension of 100. All the other sentences are represented in the same manner. All the vectorized words from a sentence are added and the total is divided by the number of words in the sentence to form a sentence embedding for a sentence.

Implementation

A sentence embedding is calculated for the first 200k sentences in the question 2 using the method described above. The final embedding for the training data has shape (n,100) where n is the total number of questions in the training data. To find a match for an input sentence, first its sentence vector is calculated using the averaging word embeddings method. Then the resulting vector is matched against all questions in question 2 and a similarity score is computed.

Evaluation

The similarity metric used for this method is the cosine similarity. Cosine similarity is a metric that determines how similar documents are regardless of size. Its formula is given by:

$$\text{Cosine Similarity} \quad = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Here A and B are two vectors. Cosine similarity computes the cosine of the angle formed by two vectors projected in a multi – dimensional space. This similarity measurement is more concerned with orientation than magnitude. The cosine similarity values range from 1 to -1. The similarity measurement is -1 if two vectors are diametrically opposed, which means they are oriented in exactly opposite directions. A similarity of 1 exists between two cosine vectors that are aligned in the same orientation, whereas if two vectors that are aligned perpendicularly, the similarity is 0. The higher the value of cosine similarity, the more similar two questions are.

Sentence Embedding with Smooth Inverse Frequency (10%)

Sentence Embedding with Smooth Inverse Frequency is based on the method presented in the paper 'A Simple but Tough-To-Beat Baseline for Sentence Embeddings' [2]. This method performs better than some deep learning methods like RNN and LSTM. In this method the weighted average of the sentence's word vectors is computed first, then the projections of the average vectors on their first singular vector is removed. Then the weight of a word w is calculated by the formula:

$$\text{Weight of a word} = \frac{\text{a}}{\text{p(w)} + a}$$

Here, a is a parameter and p(w) is the (estimated) word frequency; this is known as smooth inverse frequency (SIF). SVD is used to remove the first component from these averages and generate new sentence embeddings.

$$A = USV^T$$

Singular value decomposition is a technique for dividing a matrix A of shape mxn into three smaller matrices. Here, $U_{m \times n}$ and $V_{n \times n}$ are orthogonal vectors. $S_{n \times n}$ is a diagnol matrix. U is called the left singular vector and V is the right singular vector.

The authors describe that the process of generating a corpus can be thought of as a dynamic process in which the t-th word is produced at stage t. A vector c_t called the discourse vector, which models the topic being talked about, drives this mechanism. The likelihood of a term w being generated at time t is proportional to how close the word and the discourse are at that time t:

$$\mathbb{P}(w \text{ is emitted at time } t \mid c_t) \propto \exp(\langle c_t, v_w \rangle)$$

Here, v(w) represents the vector representation of word w. All of the c_t's in the sentence s can be replaced by a single discourse vector c_s for simplicity. The authors introduced two new parameters alpha and beta and a common discourse vector c(0) such that:

$$\mathbb{P}(w \text{ is emitted in sentence } s \mid c_s) = \alpha p(w) + (1-\alpha)\frac{\exp(\langle \tilde{c}_s, v_w \rangle)}{Z_{\tilde{c}_s}}$$
$$\text{where: } \tilde{c}_s = \beta c_0 + (1-\beta) c_s \text{ and } c_0 \perp c_s$$

The authors demonstrate using Maximum Likelihood Estimation that:

$$\tilde{c}_s = \sum_{w \in s} \frac{a}{p(w) + a} v_w$$

To get the sentence embedding, the first component of this representation is removed using Singular vector decomposition.

<u>Implementation</u>

Firstly, all the word frequencies are computed. Then, for each of the sentences, the weighted average is computed using the hyper-parameter a, which is typically set to 1e-3, and a set of pre-trained Glove word embeddings. Singular vector decomposition is calculated next using sklearn's TruncatedSVD function. Number of iterations were set to 5 which is the default value. Then the first component is removed from all sentences in the embedding matrix with column as v_s for a sentence s using the formula mentioned in [1]:

$$v_s \leftarrow v_s - uu^\top v_s$$

Here u is the first singular vector. After doing these calculations we get a sentence embedding for every question in the data. I have used the first 50k sentences for computing word embeddings as this process is time cumbersome. In a similar manner, the sentence embedding was calculated for an input sentence. Cosine similarity was used to find the top similar sentences for the input.

Evaluation Metric

To evaluate the performance of different methods performing text matching, the top 5 and top 2 matches for the first 100 questions were computed. I have used the first 50k sentences in question 2 to form the TF-IDF matrix and inverted index. For computing the average word embeddings, the first 200k sentences were used and for smooth inverse frequency 50k were used. Less number of questions were used because the process is very time cumbersome and the system crashes. Text matching was performed between the questions in question 2 list and the first 100 question 1 column. For the sentence embedding based methods, cosine similarity was calculated for every pair of questions in these two columns. The performance and results of different techniques used in this project are mentioned in results and comparisons.

Results and Comparisons

The top 2 and top 5 most similar matches were calculated for the first 100 questions in question 1 with is duplicate equal to 1, using all the different methods mentioned above. The results are shown in the table below:

Methods	**Number of training questions used**	**Text pre-processing method used**	**Accuracy top 2** (correct match found in top 2)	**Accuracy top 5** (correct match found in top 5)
TFIDF	50,000	Stop word removal, punctuation removal, stemming	17%	25%
Average Word Embedding	200,000	Stop word removal, punctuation removal,Lemmatisation	67%	77%
Smooth Inverse Frequency Embedding	50,000	Stop word removal, punctuation removal,Lemmatisation	34%	41%

Table 1: Results

The method of TFIDF uses a term-document matrix to find a matching sentence for a given sentence. The weight of a word in a document increases with the number of times it appears in that document and decreases with the number of times it appears in all documents. However, there are some disadvantages with this method due to which it had the lowest performance (25% accuracy for top 5 matches) compared to the sentence embedding based methods. One main disadvantage is that it does not capture the semantic meaning of the sentence. It does not take into account similar words and relationship between different words in a language. This problem can be solved using pre trained word embeddings.

The Glove word embedding can capture the sematic and lexical property of words. Glove embedding was used in both average word and smooth inverse frequency method. Both these methods show a better performance than TFIDF. These two methods created a sentence embedding from the word embedding using different algorithms. Average word model averages the word embeddings to form a vector sentence representation whereas the smooth inverse frequency computed the weighted average of words and modified it a little by using SVD. For the given dataset of questions, average word embedding model performed the best with 77% and 67% ground truth match for a question present in top5 and top 2 respectively.

Summary

A maximum accuracy score of 77% and 67% in top5 and top 2 matches was obtained using average word model. There is a scope of improving the model's performance further. A larger training data and further optimisation can be helpful to predict the matches correctly. Other models like RNN or LSTM can be used to achieve a better performance.

References

[1] Singh, J & Gupta, V 2016, 'Text Stemming: Approaches, Applications, and Challenges', ACM Computing Surveys, vol. 49, no. 3, pp. 1–46.

[2] Sanjeev Arora, Yingyu Liang, Tengyu Ma, ICLR 2017, 'A SIMPLE BUT TOUGH-TO-BEAT BASELINE FOR SENTENCE EMBEDDINGS'.