ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΜΑΘΗΜΑ: M161 – Data Analytics - Τεχνικές Ανάλυσης Δεδομένων Υψηλής Κλίμακας

Διδάσκοντες: κ. Δημήτριος Γουνόπουλος

Γεωργία Παρασκευοπούλου
gparaske@phs.uoa.gr

Παναγιώτης Θεοδωρόπουλος
ic1200006@di.uoa.gr

Αθήνα
Ιανουάριος 2021

Contents

Question 1a

Word clouds

Business

Figure 1

Entertainment

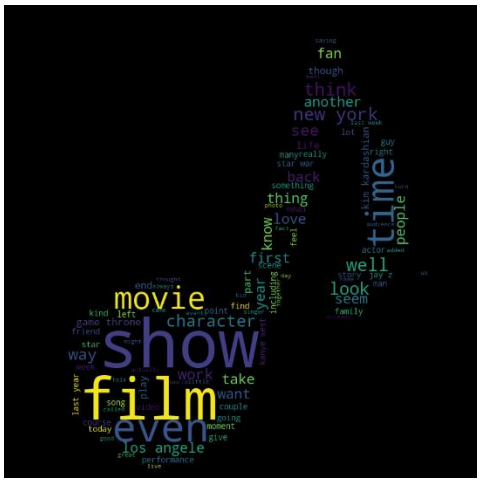


Figure 2

Health

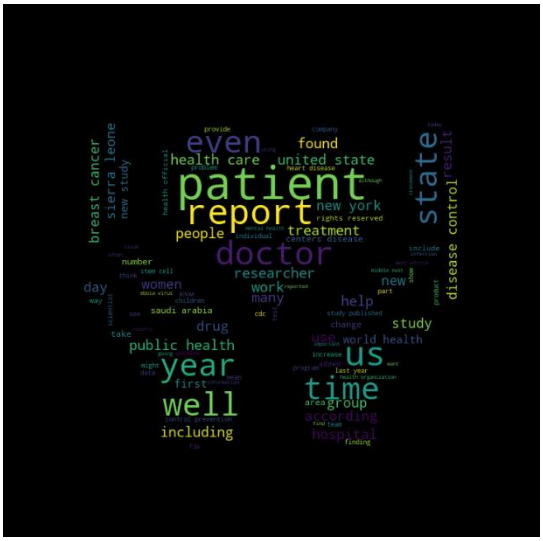


Figure 3

Technology

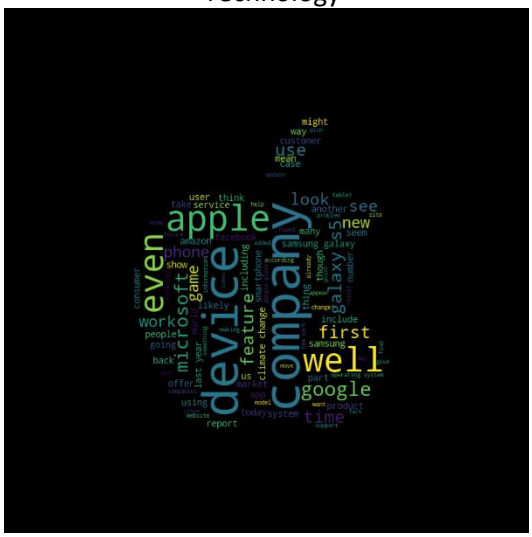


Figure 4

Cleaning

We implemented DataCleaner() class in cleanData.py. DataCleaner applies :

- removing between square brackets
- removing special characters
- removing numbers
- converting to lower-case
- removing punctuation
- removing stop words
- stemming

Example of :	**Before cleaning**	**After cleaning**
removing special characters	“a string with special characters!!!!!!”	“a string with special characters”
removing numbers	“a string with numbers 123”	“a string with numbers”
lower case	“CONVERT TO LOWER CASE”	“convert to lower case”
punctuation	“one, two, three, four”	“one two three four”
stop words	“he may say he is good at tennis”	“good tennis”
stemming	“caresses” “flies” “meeting” “references”	“caress” “fli” “meet” “refer”

For word clouds we used all cleaning operations, except stemming. We also used mask images to make word clouds with shapes related to each category. Code for this task can be found in 1a_wordCloud.py

Question 1b

Initially, we should report that in our implementation we didn't train and test our classifiers only with content of the csv file, but we also added in the title to generate more accurate recommendations using a simple concatenation:

```
X_train = df_train.iloc[:, 1] + " " + df_train.iloc[:, 2]

X_test = df_test.iloc[:, 1] + " " + df_test.iloc[:, 2]
```

TF-IDF Vectorization

In this question we firstly used the Bag of Words (BoW) approach to evaluate the models. We chose TF-IDF vectorization instead of a simple Count Vectorizer. TF-IDF stands for Term Frequency — Inverse Document Frequency. It is a statistic that defines how important a word is not only for a document, but also for other documents from the same dataset.This is performed by calculating the term frequency of word *w* in document *d* and inverse frequency which is related to the frequency of word *w* in the whole corpus. TF-IDF score is the product of these two previous frequencies and is applied to every word in every document in our dataset. As a result, for every word, the TF-IDF value increases in a logarithmic way as a function of the word frequency in a document, but gradually decreases as word's appearance in other documents is increased.

For Random Forests (RF) we tried to tune the hyperparameters of the model like Number of trees in random forest, Minimum number of samples required to split a node, Minimum number of samples required at each leaf node etc using 3-Fold Cross Validation and RandomizedSearchCV from sklearn. Unfortunately, we faced Segmentation Fault Core Dumped error after some hours of training on GPU and we didn't overcome this issue. Consequently, we finally evaluated this model for default parameters. Code for the tuning can be found in Random_forest_hyperparameter_tuning.py.

For Support Vector Machine (SVM) we used the hinge loss function and linear kernel, since hinge is the loss function for which the SVM is optimized and linear kernel has been found to outperform non-linear kernels in text classification(Rennie, 2001).

Singular Value Decomposition (SVD)

Afterwards, we should use Singular Value Decomposition (SVD) for both RF and SVM models. SVD is one of the most popular techniques for dimensionality reduction of sparse data, since using linear algebra's techniques achieves projection of data with m features into a subspace with the k most important features(k<=m), whilst retaining the essence of the original data . One important decision is how many SVD components should we use? We didn' t want to make an arbitrary choice, so we evaluated the same transform and model with different numbers of SVD components and chose the amount of dimensionality reduction that results in best average performance after 5-Fold Cross Validation.

We created a box and whisker plot for the distribution of accuracy scores for different numbers of dimensions. The range of SVD components we used is [5, 2000] (Figure 5). We observed that the more components we feed the SVM algorithm, the highest the classification accuracy . Furthermore, as we can observe after value 2000 accuracy is inclined to be stabilized. As a result, we chose the value 2000 for final

training, because the results of 2500 and 3000 were just a bit better than 2000, but with higher computational costs (slower execution time). For RF we tried range [100 , 3000](Figures 6, 7, 8) in three separate runnings due to the large required time of the process . The behaviour of Random Forest (RF) classifier verified the Hughes Phenomenon which is described as follows : "As the number of features increases, the classifier's performance increases as well until we reach the optimal number of features. Adding more features based on the same size as the training set will then degrade the classifier's performance".

It seems that value 300 of dimensions is optimal, so we chose this for final training.

You can see our implementations in 1b_TFIDF_withSVD_RF.py and 1b_TFIDF_withSVD_SVM.py.

Support Vector Machine (SVM)

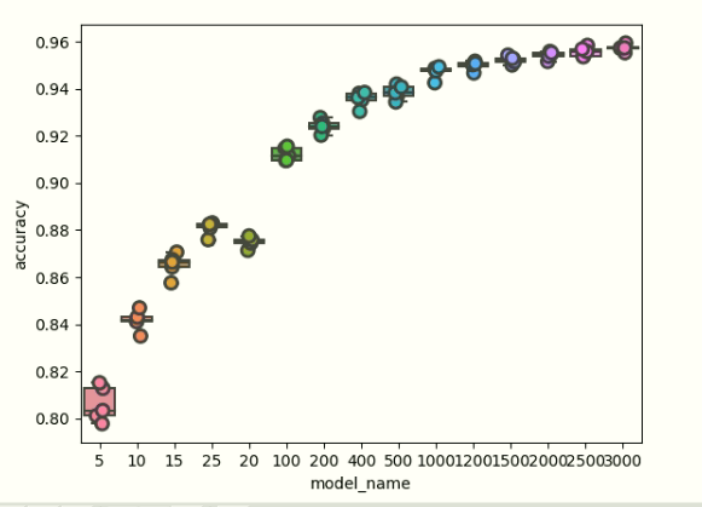


Figure 5

Random Forest (RF)

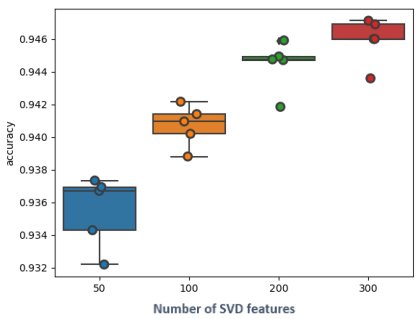


Figure 6

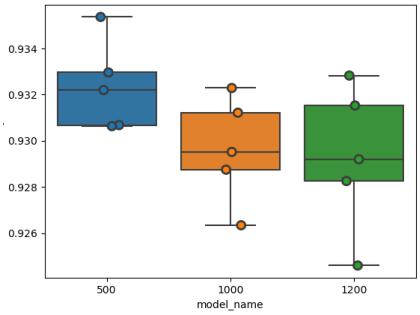


Figure 7

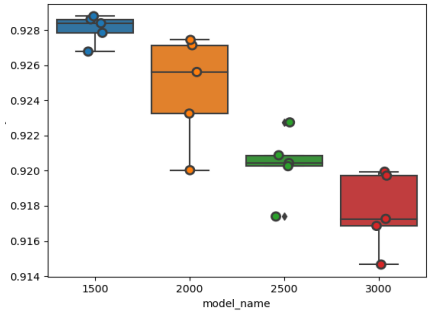


Figure 8

Final Classification Models

After choosing basic hyperparameters for our models, as we described previously, we evaluated them using 5-Fold Cross Validation. The results concluded on accuracy distributions of Figure 9.

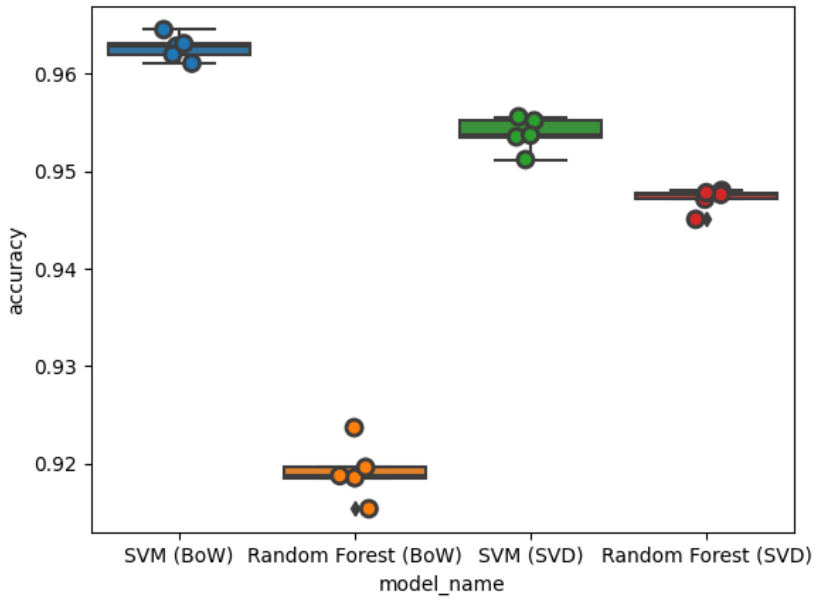


Figure 9

In order to compute recall, precision and f1 measures without using the number of examples in each class (Macro) we implemented a custom_scorer using "make_scorer" function from sklearn.metrics. This custom_scorer can be used as the value of parameter "scoring" when running cross Validation with "cross_validate" method from sklearn.model_selection. The part of code which implements this iis in method **crossValidation_and_evaluation** (..) of 1b_TFIDF_final.py :

```
custom_scorer = {'accuracy':make_scorer(accuracy_score),
                 'precision': make_scorer(precision_score, average='macro'),
                 'recall': make_scorer(recall_score, average='macro'),
                 'f1': make_scorer(f1_score, average='macro'),
                }

for model_name, model in models.items():

    scores = cross_validate(model, X, y, scoring=custom_scorer , cv=CV)
```

Method **crossValidation_and_evaluation** (..) runs for every model. Firstly implements 5-fold cross Validation and then calls method **report_and_predictions** (..) in which the current model is trained with the whole data set and evaluated using the given test set. At the end predictions for kaggle are made and saved as .csv files.

Our best model was SVM with BoW approach with accuracy about 96%.

Beat the Benchmark

Text is just a sequence of words, or more precisely, a sequence of characters. But when we vectorize texts in order to fit them to a classifier we convert them to a long list of numbers, integers or not, in order to be understood by our algorithm. TF-IDF helps our number sequences to convey the importance of a word related to the document in which it belongs and the whole dataset. Although, using only this process we don't have a lot of information about the semantic of the document, i.e context. But context is very important for text classification.This preprocess established its importance, as our results were improved.

The solution of this issue is data "cleansing". Consequently, in order to improve our best result (SVM-BoW approach) we decided to preprocess and clean our data before TF-IDF vectorization, with DataCleaner() class as at the previous question with word clouds. But now we used stemming too.

Stemming is a very useful preprocess due to its result to reduce the number of overall terms to certain "root" terms. As a consequence TF-IDF vectorization gives different sequences of numbers after stemming application. It's worth to say that the process of stemming could affix words containing additional information, which can be utilized. For example, "faster" and "fastest" have the same root, but their semantic meaning is different from each other. To be sure of our decision to include or not include stemming we tried both. Our best results were when including stemming and these we will report.

Another very important cleansing operation for elevation of semantic meaning is "removing stopwords". Stopwords are the words which are used very frequently, and they're so frequent, that they somewhat lose their semantic meaning. Words like "of, are, the, it, is" are some examples. As a result for document classification, where keywords are more important than general terms, removing stopwords is a good idea.We implemented this by removing the list of every documents' tokens that belong to the predetermined english stopword list of nltk.corpus.

For training of this mode of SVM we ran 1b_SVM_cleanData.py. The accuracy distribution can be seen at the following box and whisker plot (Figure 10). The accuracy increased a little compared to simple SVM.

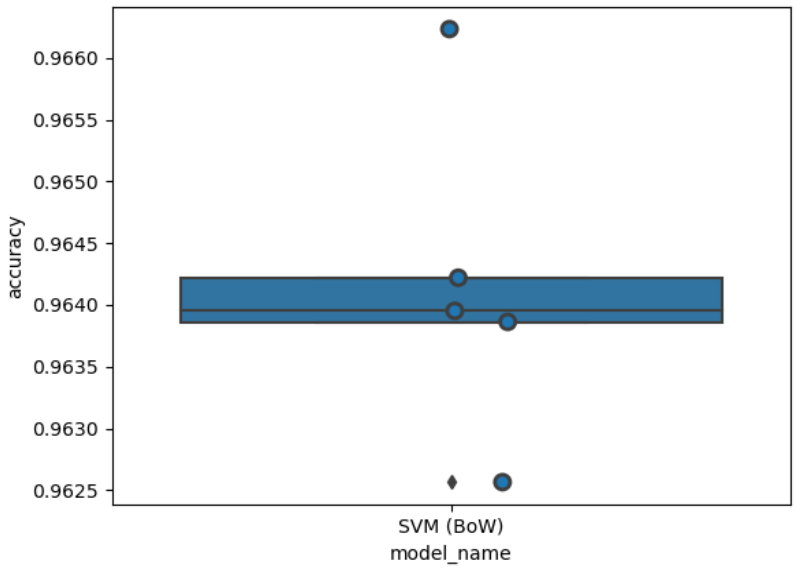


Figure 10

Classification Results

Results of all our experiments are summed up to the following table.

Statistic Measure	SVM (BoW)	Random Forests (BoW)	SVM (SVD)	Random Forests (SVD)	My method (SVM – BOW) Cleaned data
Accuracy	0.962709	0.919209	0.953844	0.947135	0.964167
Precision	0.962996	0.926091	0.953790	0.949883	0.964044
Recall	0.955744	0.897793	0.944336	0.932804	0.957872
F-Measure	0.959235	0.910465	0.948870	0.940783	0.960853

As we can see SVD helped Random Forests method improve its scores about 3%. In contrast, we cannot see the same behaviour with Support Vector machines. SVM has better scores for the BoW approach and improved results after use of cleansing techniques.

It' s worth to mention that we experimented with extra methods like Naive Bayes and a Bi-directional LSTM for text classification, but we didn't improve our results. For Bi-LSTM we tried to tune the hyperparameters for the model but we couldn' t beat the accuracy of SVM.

Question 2a

Table with results

Type	BuildTime (sec)	QueryTime (sec)	TotalTime (sec)	#Duplicates (#test questions in train)	Parameters
Exact-Jaccard	0	4013.28	4013.28	572	word-shingles
Exact-Jaccard	0	29529.68	29529.68	402	3-shingles
Exact-Jaccard	0	29887.71	29887.71	280	4-shingles
Exact-Jaccard	0	12912.93	12912.93	206	5-shingles
LSH-Jaccard	502.12	0.06	502.18	810	word-shingles,16 perm
LSH-Jaccard	755.07	0.15	755.22	624	word-shingles, 32 perm
LSH-Jaccard	1049.11	0.14	1049.25	681	word-shingles, 64 perm
LSH-Jaccard	4455.80	0.76	4456.56	569	3-shingles, 16 perm
LSH-Jaccard	2968.96	0.89	2969.85	420	3-shingles, 32 perm
LSH-Jaccard	2984.37	1.13	2985.50	460	3-shingles, 64 perm
LSH-Jaccard	2236.78	0.85	2237.63	392	4-shingles, 16 perm
LSH-Jaccard	2714.18	1.26	2715.39	293	4-shingles, 32 perm
LSH-Jaccard	4733.04	1.63	4734.67	341	4-shingles, 64 perm
LSH-Jaccard	3550.71	1.09	3551.80	295	5-shingles, 16 perm
LSH-Jaccard	4049.44	1.38	4050.82	239	5-shingles, 32 perm
LSH-Jaccard	5049.73	1.21	5050.94	262	5-shingles, 64 perm

Jaccard Similarity

The Jaccard similarity (coefficient) is defined as follows, given two word sets A and B:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The numerator $|A \cap B|$ expresses the multitude the size of the intersection of A and B (Figure 11)

The denominator $|A \cup B|$ expresses the size of union of A and B (Figure 12)

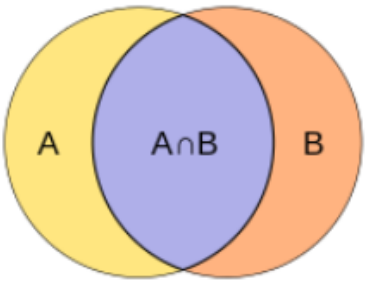


Figure 11

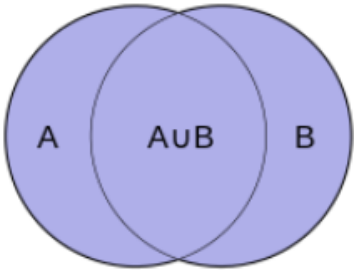


Figure 12

We implemented Jaccard Similarity according to the previous definition, testing every question of the test set with every question of train set:

```
def jaccard(set_a, set_b):
    intersection = set_a & set_b
    union = set_a | set_b
    return len(intersection) / len(union)
```

We tried different methods for representing documents as sets. For the purpose of identifying lexically similar documents we firstly experimented with k-shingles. A k -shingle is defined as any substring of length k found within the document . Then, we associate with each document the set of k-shingles that appear one or more times within that document. If we do so, then documents that share pieces will have many common elements in their sets, even if those sentences appear in different orders in the two documents. For example, the Jaccard similarity of words “Nadal” and ”Nadia” using 2-shingles J= 2/6 = 1/3 (Figure 13)

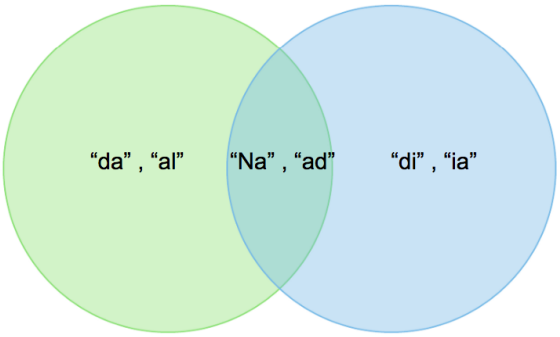


Figure 13

We experimented with k=3,4,5. We also tried an alternative form of shingling, word shingles (tokens) after removing special characters from questions. Functions for set representation are:

```
def get_shingles(text, char_ngram):
    return set(text[head:head + char_ngram] for head in range(0, len(text) - char_ngram))
```

(Full code for Jaccard with k-shingles is available in 2a_exact_Jaccard_k_shingles.py)

```
def set_representation(data):
    shingles=[]
    for i,question in enumerate(data):
        word_list = []
        question=remove_special_characters(question)
        tokenizer=ToktokTokenizer()
        words = tokenizer.tokenize(question)
```

```
    for word in words:
        word_list.append(word.lower())
    shingles.append(set(word_list))

  return shingles
```

(Full code for Jaccard with word shingles is available in 2a_exact_Jaccard_word_shingles.py) .

Let's see a specific pair of questions :

Q1 : "What are some things that doctors know, but most people don't?"

Q2 : "What are some things that most people don't know are stupid?"

5-shingles sets

Set(Q2) : A={' peop', "'t kn", 'w are', 'gs th', 'e thi', ' that', ' most', 'le do', 'tupid', 'are s', 're st', 'ost p', 'hings', " don'", 'peopl', "on't ", 'know ', 'ow ar', ' some', 'some ', ' know', 're so', 'ople ', 'ple d', "don't", 't kno', 'e don', 'me th', 'now a', 'ings ', 'st pe', 'e stu', 'most ', ' stup', 'what ', 'ome t', ' thin', ' are ', 'that ', 'hat m', 't peo', 'at mo', 't mos', 'ngs t', 'at ar', 'e som', "n't k", 'upid?', 'stupi', 'hat a', 's tha', 't are', 'eople', 'thing'}

set(Q2) : B= {'n\'t?"', '"what', ' peop', 'gs th', 'e thi', ' that', ' most', 'le do', 'are s', 'ost p', 'hings', 'know,', " don'", 'peopl', ' some', 'ctors', 'some ', 'ors k', ' know', 're so', 'ople ', 'ple d', "don't", ' but ', 'e don', 'w, bu', ', but', 'me th', 't doc', 'ings ', 'st pe', 's kno', 'hat d', 'most ', 'what ', 'now, ', 'rs kn', 'ome t', ' thin', ' are ', 'that ', 'at do', 'docto', 'octor', 'ut mo', 't peo', 'but m', 't mos', 'ngs t', 'at ar', 'e som', "on't?", 'ow, b', 'hat a', 'tors ', 's tha', ' doct', 't are', 'eople', 'thing'}

$|A \cap B| = 38$, $|A \cup B| = 76$ so Jaccard similarity is equal to J(A,B) = $\frac{38}{76}$=0.5

word-shingles sets

Set(Q2) : A={'are', 'some', 'most', 'that', 'know', 'dont', 'what', 'people', 'but', 'things', 'doctors'}

set(Q2) : B= {'are', 'some', 'that', 'dont', 'know', 'stupid', 'what', 'people', 'things', 'most'}

$A \cup B$ ={'know', 'are', 'what', 'people', 'some', 'doctors', 'but', 'that', 'dont', 'stupid', 'most', 'things'}

$A \cap B$ ={'know', 'people', 'are', 'some', 'that', 'dont', 'what', 'most', 'things'}

$|A \cap B| = 9$, $|A \cup B| = 12$ so Jaccard similarity is equal to J(A,B) = $\frac{9}{12}$=0.75

We also report that we have found some missing questions in both corpusTest.csv and corpusTrain.csv files, and we decided to skip the rows with missing questions. The similarity for these questions is 0 (zero).

Cosine Theory

The Cosine similarity is defined as follows, given two vectors A and B:

$$C(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Where n is the number of vectors' dimensions.

We firstly implemented this definition as follows (See in 2a_exact_cosine.py) :

```
def cosine(vector1 , vector2):
    print(np.dot(vector1 , vector2))
    if(np.linalg.norm(vector1)==0 or np.linalg.norm(vector2)==0):
```

```
    cosine = 0.0
  else:
    print(np.linalg.norm(vector1)*np.linalg.norm(vector2))
    cosine = np.dot(vector1 , vector2)/(np.linalg.norm(vector1)*np.linalg.norm(vector2))
  return cosine
```

In this case we translate questions to vectors using CountVectorizer(max_features=10000). Afterwards, we tried to test every question of the test set with every question of the train set, but memory error arised. Then, we tried to see the problem from a different view. We thought about using linear algebra and making computations of matrices.

```
def cosine(matrix1 , matrix2):
  norm1=np.sum(np.abs(matrix1)**2,axis=-1)**(1./2)
  norm2=np.sum(np.abs(matrix2)**2,axis=-1)**(1./2)
  norm1 = norm1.reshape(len(norm1),1)
  cosine_matrix = np.dot(matrix1 , np.transpose(matrix2))/(norm1*norm2)
  return cosine_matrix
```

Here, we faced a sparse matrix problem. To overcome this issue we decide to use **cosine_similarity** from sklearn.metrics.pairwise , a ready-made python function that uses sparse matrices,instead of our implementation (2a_exact_cosine_final_version.py).

As you can see from the table with results the above two methods found had differences at execution times. Reduced execution time in Exact-Cosine is explained because we looked at the whole test with every element of the train. Plus **cosine_similarity** works quite fast. While, in the Exact-Jaccard we examine data all one by one.

Two methods also found quite different number of duplicates. Differences in the number of duplicates are observed and between different approaches of shingling. Let's explain these results by analyzing some exact examples.

Examples of Similarity Results

Example 1

Training Question with **ID=313502**	*What are the RTO formalities to transfer a car from Pune (MH 12 passing) to Chennai?*	***Explanation:** Here we observe that all of the metrics are beyond 0.8 threshold except 3-gram shingles. This happens because the 3-shingles method is very sensitive. Chennai includes 7 characters and Bangalore 10. This little difference results in the number of every word's set of shingles, by extension, the number of their union and intersection.*
Test Question with **ID = 5**	*What are the RTO formalities to transfer a car from Pune (MH 12 passing) to Bangalore?*	
Jaccard Similarity (word - shingles)	*0.875*	
Jaccard Similarity (3 - shingles)	*0.79*	
Jaccard Similarity (4- shingles)	*0.8*	
Jaccard Similarity (5 - shingles)	*0.8*	
Cosine Similarity	*0.937*	

Example 2

Training Question with **ID=5361**	"Which is a suitable inpatient drug and alcohol rehab center near Greene County GA?"	***Explanation:*** This pair is counted as duplicate only by Jaccard with word - shingles and Cosine similarity. "Greene" includes 6 characters and "Meriwether" 4 more. This 4 char difference affected all k-singles methods. Although, word-shingles method stayed unaffected because it counts only tokens i.e. 1 token difference. But, why is cosine similarity 1? Count Vectorizer creates a vocabulary from the 10.000 most frequent words of the corpus. "Greece" and "Meriwether '' are not in the vocabulary. So these two questions has the same vector representation and as a result cosine similarity of the same vectors are 1.
Test Question with **ID =51**	"Which is a suitable inpatient drug and alcohol rehab center near Meriwether County GA?"	
Jaccard Similarity (word - shingles)	*0.866*	
Jaccard Similarity (3 - shingles)	*0.788*	
Jaccard Similarity (4- shingles)	*0.760*	
Jaccard Similarity (5 - shingles)	*0.739*	
Cosine Similarity	*1*	

Example 3

Training Question with **ID=149566**	"What are some things that doctors know, but most people don't?"	***Explanation:*** This pair's questions are quite different. Jaccard similarity with word-shingles is 0.75 and Cosine Similarity 0.836. Tokenization and Count Vectorization don't take into account the word order. As a consequence, these two questions are considered as questions with only two word differences.
Test Question with **ID =5089**	"What are some things that most people don't know are stupid?	
Jaccard Similarity (word - shingles)	*0.75*	
Jaccard Similarity (3 - shingles)	*0.6*	
Jaccard Similarity (4-shingles)	*0.547*	
Jaccard Similarity (5 - shingles)	*0.5*	
Cosine Similarity	*0.836*	

Example 4

Training Question with **ID=328600**	"How can we find professors, PhD & masters students from universities of Malaysia who would like to share their knowledge for free to record MOOCs?"	***Explanation:*** "Malaysia" has only 2 more characters than "France". As a result all of our approaches found this pair as duplicate. It is worth to comment that as the number of shingles decreases, the Jaccard Similarity also increases. However, if someone uses k = 1, for a Deduplication task, most documents will have most of the common characters and few other characters, so almost all documents will have high similarity. So, the optimal number of k is large enough that the probability of any given shingle appearing in any given document is low.
Test Question with **ID =5087**	"How can we find professors, PhD & masters students from universities of France who would like to share their knowledge for free to record MOOCs?"	
Jaccard Similarity (word - shingles)	*0.916*	
Jaccard Similarity (3 - shingles)	*0.889*	
Jaccard Similarity (4- shingles)	*0.868*	

Jaccard Similarity (5 - shingles)	*0.856*	For our Question De-Duplication task we think that 3-shingles is a good choice.
Cosine Similarity	*0.961*	

MinHash LSH

It's been shown earlier that the Jaccard Similarity can be a good string metric, however, we need to split each question into the words, then, compare the two sets, and repeat for every pair. But as the amount of questions grows, the number of comparisons also grows, but quadratically. The same happens with execution time.

LSH stands for Locality Sensitive Hashing, and it is a technique that is used for finding similarities between items, by creating a simple fixed-size numeric fingerprint for each sentence and then just comparing the fingerprints, as depicted in the Figure 14 (Shikhar Gupta) below:

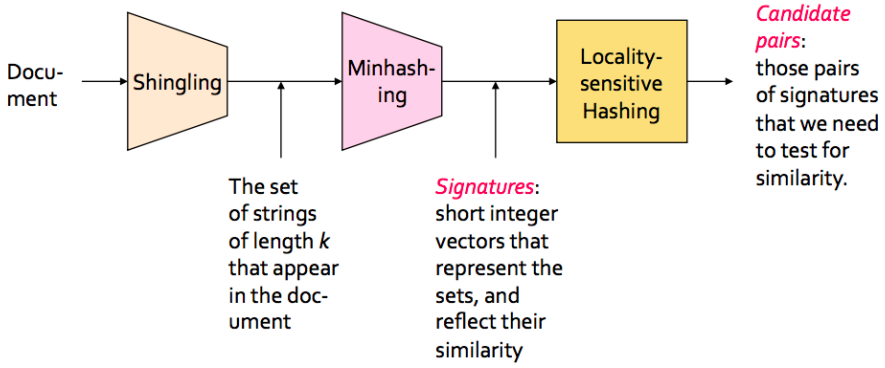


Figure 14

We chose datasketch in order to find how many of the documents in the test-set already exist in the train set using Jaccard LSH. This package includes python functions which accomplish Min-hash LSH efficiently. According to datasketch documentation if we create a MinHash for every set, when a query comes, Jaccard similarities between the query MinHash and all the MinHash of the collection will be computed. Then sets that satisfy our threshold will be returned.

Main function of our implementation is **find_number_of_duplicates_between_sets**(train, test, threshold, kind_of_shingles, no_of_permutations) .

threshold : Jaccard similarity threshold which will be used as a parameter in MinHashLSH

no_of_permutations : the number of permutations we want for the MinHash algorithm. The higher the permutations the longer the runtime.

kind_of_shingles : integer for k-shingles or "word" for word shingles

In this function ,next operations take part in:

- ***Set Representation**:* Depending on the value of kind_of_shingles is integer or "word" **set_k_shingles_representation**(data, kind_of_shingles) or **set_word_shingles_representation**(data)is called respectively. In both of those functions, set_dict maps question id (eg 'm328600') to set representation of question and norm_dict maps question id (eg 'm328600') to actual question. We loop through each question, convert them into shingles k-shingles or word shingles respectively. For word shingles removing special characters operation is used.

- ***Minhasing:*** Minhashing in principle is the representation of a column of the characteristic matrix in a fixed-size numeric fingerprint as follows. We pick a permutation of the rows. The minhash value of any column is the number of the first row, in the permuted order, in which the column has a 1. For that purpose, in our implementation **create_minHash_signatures**(dictionary, no_of_permutations) is called for train and test corpus.

```
def create_minHash_signatures(set_dict, no_of_permutations):

        min_dict = {}
        count = 0
        for val in set_dict.values():
            m = MinHash(num_perm=no_of_permutations)
            for shingle in val:
                m.update(shingle.encode('utf8'))
            min_dict["m{}".format(count)] = m
            count+=1
        return min_dict
```

min_dict maps question id (eg 'm328600') to min hash signatures. We loop through all the set representations of questions and calculate the signatures and store them in the min_dict dictionary. It's worth saying that the update function of MinHash uses the SHA1 hash function, which requires bytes as input.

- **Locality Sensitive Hashing** : We set the Jaccard similarity threshold as a parameter in MinHashLSH. We loop through the signatures or keys in the min_dict dictionary.

```
lsh = MinHashLSH(threshold, no_of_permutations)
for key in min_dict.keys():
        lsh.insert(key,min_dict_train[key])
```

- **Search for duplicates :** We call **calculate_similarities**(min_dict_test , lsh) which using lsh.query() saves as a list of tuples like (2133,['m379827', 'm262565']). First element in tuple depicts the query ID from the test set and the second is a list with the values of all the questions that seem similar based on the threshold.

```
def calculate_similarities(dictionary ,lsh):
    number_of_duplicates=0
    test_question_exist_inTrain=0
    duplicates=[]
    for i ,  query in enumerate(dictionary .keys()):
        bucket = lsh.query(dictionary [query])
        duplicates.append((i,bucket))
        if (len(bucket)>0):
            test_question_exist_inTrain+=1
        return duplicates ,  test_question_exist_inTrain
```

We ran the previous functions for 3, 4, 5 and word shingles. At this point it is very important to note that according to datasketch documentation the lsh.query does not give us the exact duplicates, due to the use of MinHash and LSH. If we have a look in the Source code for datasketch.lsh we see that optimal parameters for number of bands and number of rows per band are chosen through minimization of false positives and false negatives. As a result, the list with the values of all the questions that seem similar based on the threshold will include false positives - sets that do not satisfy the threshold but returned, and false negatives - qualifying sets that are not returned. For that reason, we observe an increased number of duplicates at all LSH-Jaccard runs compared to Exact Jaccard results. However, the property of LSH assures that sets with higher Jaccard similarities always have higher probabilities to get returned than

sets with lower similarities. This is based on the following remarkable connection between minhashing and Jaccard similarity of the sets that are minashed according to Chapter 3, Mining of Massive Datasets: "The probability that the minhash function for a random permutation of rows produces the same value for two sets equals the Jaccard similarity of those sets."

Full code is available in 2a_minHash_Jaccard.py

Cosine LSH

Let suppose$\vec{u}$, a word vector of d dimensional space which represents a document. We then suppose a vector $\underline{r}$ of d dimensional space, normal to α random hyperplane. This random vector represents the random hyperplane which cuts the dimensional space into two regions. Afterwards, we compute the dot product $\vec{u} \cdot \vec{r}$. If $\vec{u} \cdot \vec{r} \geq 0$ we will assign $\underline{u}$ to 1. In contrast, if $\vec{u} \cdot \vec{r} < 0$ we will assign $\vec{u}$to 0. Each hyperplane is the set of points whose dot product with $\underline{r}$ is 0 and is seen as representing a hash function, whose value applied to a document is either 0 or 1. For planes drawn at random, it can be mathematically proven that two word vectors $\vec{x}$ and $\vec{y}$ hash to the same value with probability:

$$P[h(x) = h(y)] = 1 - \frac{\theta}{\pi}$$, where θ is the angle between $\vec{x}$ and $\vec{y}$.

So, the closer two documents are (small θ) the greater the chances they end up in the same region and as a result with the same hash 0 or 1.

Now, given a train corpus and a test corpus with the purpose to identify duplicate documents we work as follows :

- Instead of using only one hash function from the hyperspace to the space {0,1} we apply (to every document) k x L distinct hash functions i.e. we use L sets of k random vectors
- We divide this k x L hash functions in L bands/groups of k. In every group we produce a k bit signature of every document. This signature is a projection of the word vector, which usually has huge dimensions, to k dimensions.
- A pair of documents is considered a candidate pair, if and only if they have the same values on all the hash functions in a band for at least 1 band. (Figure 15).
- We examine each candidate pair for cosine similarity above chosen threshold.

		d^1	d^2	d^3	d^4	.....	d^N
band 1	h_1	1	1	0	1	...	1
	h_2	0	0	1	1	...	0
	h_3	1	0	1	0	...	0
band 2	h_4	1	1	1	0	...	1
	h_5	0	0	0	0	...	1
	h_6	0	1	0	0	...	1
band 3	h_7	1	1	0	0	...	1
	h_8	1	0	1	0	...	0
	h_9	0	1	1	0	...	1
	h_{10}	1	1	0	1	...	1
⋮	....	...	...	...	...	...	...
	h_{k*L}	1	1	0	1	...	0

Figure 15

Two first steps of the previous process are visualized at Figure 16. There, we introduce an example where vectors are in 10000 dimensional space and k=5.

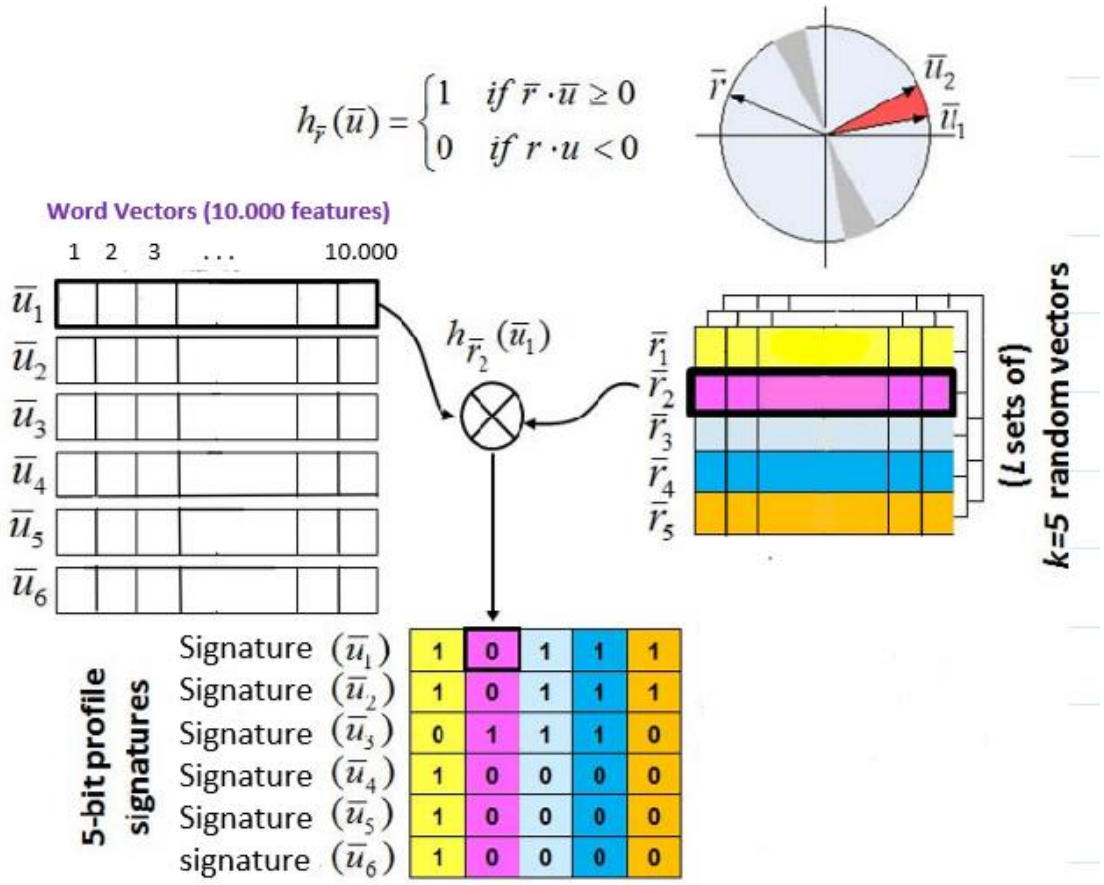


Figure 16

In this question we used TF-IDF Vectorizer (lowercase=True, max_features=10000) to vectorize the Quora questions of test and train corpus. Our implementation can be found in 2a_LSH_cosine.py which includes three classes :

- sameQuestionSearch(num_tables, hash_size, inp_dimensions, training_files)
- LSH(num_tables, hash_size, inp_dimensions)
- HashTable(hash_size, inp_dimensions)

Let's analyse every class extensively.

class **HashTable** (hash_size, inp_dimensions)

Creates an object with k random vectors and a dictionary with hashes of input vectors

Parameters :

- hash_size *(int)* - Number of random hyperplanes i.e number of random vectors r
- inp_dimensions *(int)* - Dimension of input vectors u

__init__(hash_size, inp_dimensions)

Initialize self.

self.hash_size = hash_size

self.hash_table = dict()

self.projections is the table of size hash_size x inp_dimensions

generate_hash(inp_vector)

Generates hash(signature) of input vector $\underline{u}$

Parameters : inp_vector - input vector

__setitem__(inp_vec, label)

Fills in hash_table dictionary storing hash value keys in labels . Label should be the ID of the question.

__getitem__(inp_vec)

Generates a hash of input vector. Returns the label of that hash in the hash_table.

class LSH (num_tables, hash_size, inp_dimensions)

Creates an object with L sets of k random vectors i.e L objects of HashTable class

Parameters :

- num_tables (int) - Number L of sets or random vectors
- hash_size *(int)* - Number of random hyperplanes i.e number of random vectors $\underline{r}$
- inp_dimensions *(int)* - Dimension of input vectors $\underline{u}$

__init__(num_tables, hash_size, inp_dimensions)

Initialize self.

self.hash_size = hash_size

self.hash_tables : list of HashTable Objects

__setitem__(inp_vec, label)

Loops through hash_tables. Fills in every hash_table dictionary storing hash value keys in labels. Label should be the ID of the question.

__getitem__(inp_vec)

Loops through L hash_tables. For every table a list is extended with the label of question with same hash of input vector. Returns the list with the labels with the same values with input vector.

class sameQuestionSearch (num_tables, hash_size, inp_dimensions, training_files)

Creates an object with L sets of k random vectors i.e object of LSH class. Fill in the hashTables with the hashes of training files. Given a set of queries searches through hash tables for same hashes between queries and training files. A pair of documents is considered a candidate pair, if and only if they have the same hash value on a hashTable for at least 1 hashTable of the L hashTables. Then, each candidate pair is examined for cosine similarity above chosen threshold.

Parameters :

- num_tables (int) - Number L of sets or random vectors
- hash_size *(int)* - Number of random hyperplanes i.e number of random vectors $\underline{r}$

- inp_dimensions *(int)* - Dimension of input vectors $\underline{u}$
- training_files (list) - List of vectorised questions for training

__init__(num_tables, hash_size, inp_dimensions, training_files)

Initialize self.

self.lsh : LSH Object

train()

Loops through the training files . Fills in every hash_table dictionary of self.lsh. ID of every training file is given as the label parameter.

find_similar_question(question, candidates_array)

Parameters :

- question (array) - Vectorized question for testing
- candidates_array (array) - array of candidate questions with same hash with question through L hashTables. It is in the form that is returned from LSH. __getitem__(inp_vec) and it may include some candidates more than one time.

  Find the unique elements of candidates_array. Compute the cosine_similarity matrix of question and every candidate. Returns the candidates that have cosine similarity beyond 0.8.

query(self, questions)

Parameters:

- questions (list) - List of vectorised questions for testing

  Loops through the list of questions and finds real duplicates calling self.find_similar_question()

  Saves at csv file the list with tuples. Tuples are like (2133,[379827, 262565]). First element in tuple depicts the query ID from the test set and the second is a list with the values of all the questions that had the same hash with the query and seem similar based on the threshold.

It's important to report that the list with the values of all the questions that seem similar based on the same hash will include false negatives. Some questions don't have the chance to be candidates due to the stochastic nature of LSH. The random vectors may be chosen with a way that some vectors, with similarity beyond 0.8, will not have the same hash. Although, this implementation has false positives because of the threshold check in **find_similar_question**(..) method.

We run the previous method for k in range [1,10] and L=1. As we can observe from the above results as the k is increased the number of duplicates are decreased. This result comes out because as k is increased the number of false negatives is increased too, as it is more difficult for two questions to have the same hash (signature of k bits). For k=1 duplicates approximate the number of duplicates from exact cosine, but the total time that it is required is very large.

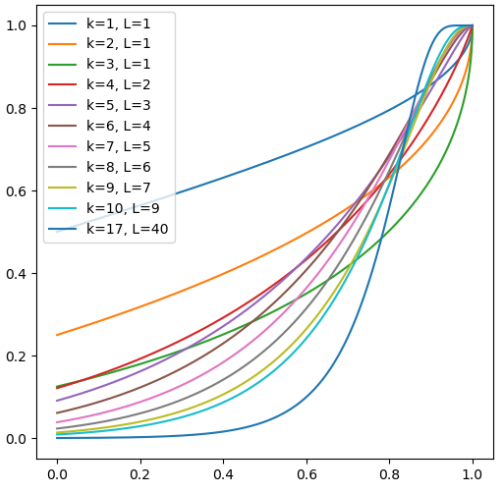
Exact-Cosine	0	2281.53	2281.53	1625	-
LSH-Cosine	1777.63	1408.13	3185.76	1394	k=1
LSH-Cosine	913.15	660.03	1573.18	1264	k=2
LSH-Cosine	612.41	404.14	1016.55	1136	k=3
LSH-Cosine	353.98	189.49	543.47	1072	k=4
LSH-Cosine	245.12	96.37	341.49	969	k=5
LSH-Cosine	254.16	93.87	348.03	905	k=6
LSH-Cosine	344.93	59.02	403.95	838	k=7
LSH-Cosine	339.03	46.75	385.78	786	k=8
LSH-Cosine	345.96	41.55	387.51	676	k=9
LSH-Cosine	331.50	35.03	366.53	644	k=10
LSH-Cosine	12313.80	254.39	12568.19	1454	k=17, L=40

The truth is that the choice of k and L is of great importance. Now the probability that two documents are considered similar pairs is:

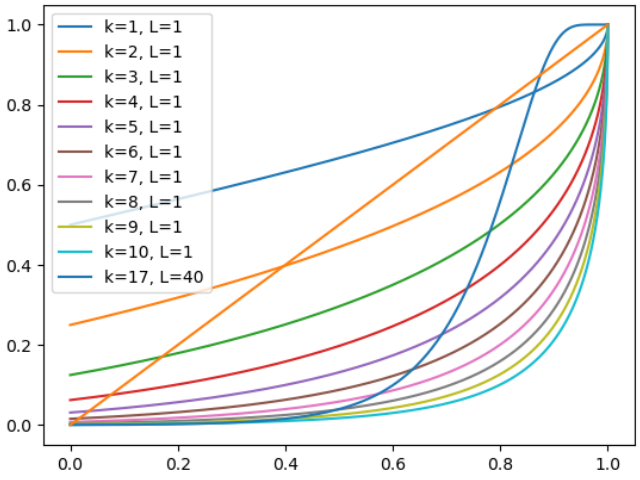
$$P[h(x) = h(y)] = 1 - (1 - p^k)^L, \text{ where } p = 1 - \frac{arccos(sim(x,y))}{\pi}$$

We plot this probability depending on the similarity of x and y for different k, L values (Figure 17 and 18, See code in lsh_cosine_probabilities.py). According to Chapter 3, Mining of Massive Datasets threshold is equal to $(1/L)^{1/k}$. In this case the curve of P should be like sigmoid and threshold the value of similarity s at which the probability of becoming a candidate is ½. Then false positives and false negatives are minimised.

From the next plots we see that k=17, L=40 is the best approximation for threshold 0.8. We tried our method for this combination and our false negatives really were minimised, as it was expected. Method found almost as duplicates as exact cosine. The build time was very large, but the query time was very small compared to the exact cosine's total time. This would be very useful for an application, which could be trained once and then use the hashTables to find duplicates of a new query fast.



x - axis : sim(x,y) , y - axis : P
Figure 17



x - axis : sim(x,y) , y - axis : P
Figure 18

Question 2b

For this question we firstly experimented with our data using 2b_data_experimentation.py. We first plotted the number of pairs that were Duplicates and the number of pairs that were not (Figure 19). As can see there is an imbalance between training class. Then we plotted the question length distribution, and we observed that most questions had length about 50 (Figure 20).

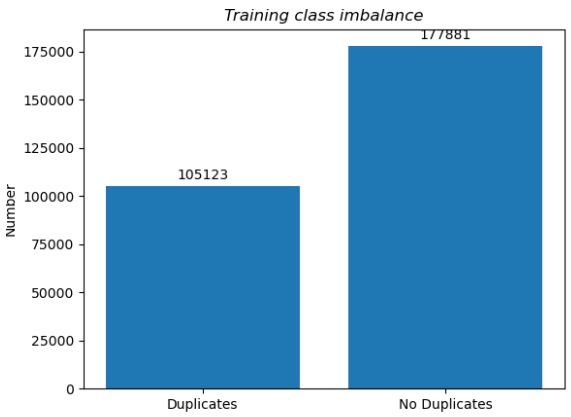


Figure 19

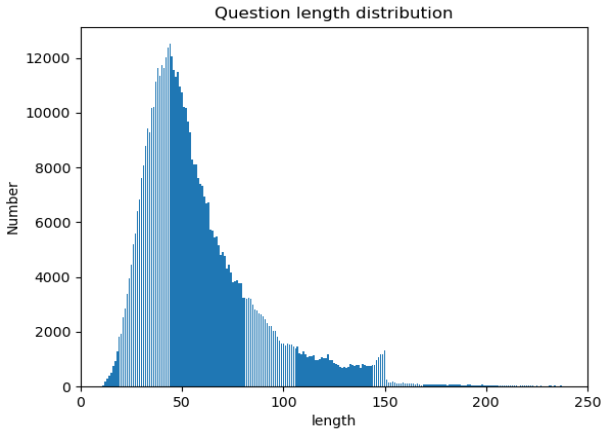


Figure 20

Afterwards we checked if there are any rows with null values :

```
nan_rows = corpus_raw[corpus_raw.isnull().any(1)]
print (nan_rows)
```

We found two pairs with null rows in the train set and one in the test set. We introduce the result for training set :

	Id	Question1	Question2	IsDuplicate
105780	105780	How can I develop android app?	NaN	0
201841	201841	How can I create an Android app?	NaN	0

Then we filled the null values with " " :

```
corpus_raw = corpus_raw.fillna('')
nan_rows = corpus_raw[corpus_raw.isnull().any(1)]
print (nan_rows)
```

To answer this question, we had to understand when two questions are believed as duplicates. According to Bogdonova et al. (2015) two questions are semantically equivalent if they can be answered by the exact same answers. Recently, deep learning methods have made significant gains in tasks like semantic equivalence detection, surpassing traditional machine learning that use hand-crafted feature extraction techniques. We decided to explore different configurations of deep neural networks to identify duplicate pairs of questions. Most deep learning methods for detecting semantic equivalence rely on a "Siamese" neural network architecture that takes two input sentences and encodes them individually using the same neural network. The resulting two output vectors are then concatenated and are given as input to a dense output layer or a shared neural network. Bowman et al. (2015) used a stack of three 200d tanh layers, with the bottom layer taking the concatenated sentence representations as input and the top layer feeding a softmax classifier. Another choice is to compare the two output vectors using some distance

metric. This last approach is used successfully by both Bogdanova et al. (2015) with cosine similarity as distance metric. Homma et al. (2016) used a concatenated vector instead of a single distance metric.

We would like to experiment these techniques. We chose as Method 1 a Siamese followed by a deep network with fully connected layers and as Method 2 a Siamese followed by a concatenated vector of two distance metrics. Two methods had one more difference. Method 1 used GloVe 300d pre-trained word Embeddings, where Method 2 used Word2Vec.

Both word2vec and GloVe enable us to represent a word as a vector that encodes the meaning of and relationship between the words. Using simple mathematics, it can be determined if two words are similar in meaning or completely opposite. This happens due to mapping human language into a geometric space in a way that semantically similar words are close together. Additionally, geometric distance between any two word vectors relate the semantic distance between the associated words. For example, king-queen and man-woman should have the same distance in embedding space. Lastly, arithmetic can be used on embeddings to derive meaning. As a result, when we use Word2Vec or Glove, documents that may not have common words, their semantic similarity can be captured. Both Word2Vec and Glove derive relationships between words using cosine distance. The two models differ in the way they are trained, and hence lead to word vectors with subtly different properties.

Data Preprocessing

Both Method-1 and Method-2 preprocess the questions in four steps:

1. **Data Cleansing**
   Cleaning functionality is implemented by the function **def Data_Cleaning(data)**.
2. **Sentence Tokenization**
   A large dictionary is created that maps each word to a unique integer index. Then the dictionary is used to convert sentences from sequences of strings to sequences of integers.
3. **Zero Padding**
   Ensures that the input to the neural network model is of uniform length. So, the max length used is **25**. If a question has more than 25. It is truncated, if it is less than **25**, zeroes are inserted to match the 25-length requirement.
4. **Embedding matrix**
   The last step uses a pre-trained word embedding to convert each word into a vector representation. Each word is converted into a **300** long vector.

Therefore, a **tensor** is created from the text data, which has dimensions **(282004, 25, 300)**, for each Question 1 and Question 2.

General Approach

Both Method-1 and Method-2 include method **crossValidation**(..). From its name it is obvious that implements k - fold cross Validation as follows:

```
kf = KFold(n_splits=splits, random_state=0, shuffle=False)

...

  for train_index, test_index in kf.split(y):

    train_questions, test_questions = np.array(X)[train_index], np.array(X)[test_index]

    train_labels , test_labels = np.array(y)[train_index], np.array(y)[test_index]
```

For each fold :

- model is built after calling the appropriate function **build_siamese_deep_net**(),for Model-1, or **build_siamese**(), for Model-2. Keras deep learning framework is used.
- model is trained for 25 epochs with early stop callback and is evaluated calculating accuracy, recall, precision and f1 metrics. See **fit_and_evaluate_model**(..) function.
- plots accuracy and loss curves (Figures 23 and 24) in order diagnose problems of our models and choose the appropriate number of epochs that we will train our final models with all data for making kaggle predictions.

After running cross Validation for each Method we saw that 10 epochs are enough for learning. Afterwards we used **fit_all_data_predict_ublabeled**(..) for training our final models for 10 epochs, fitting them with all data and predicting the Kaggle data questions pairs.

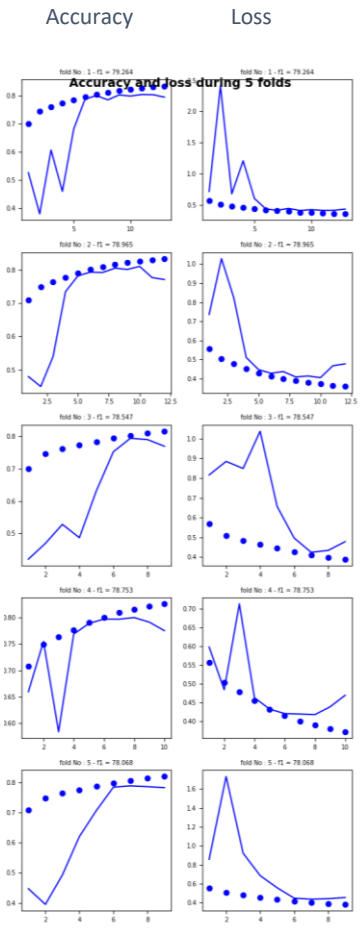


Figure 23

[Method -1]

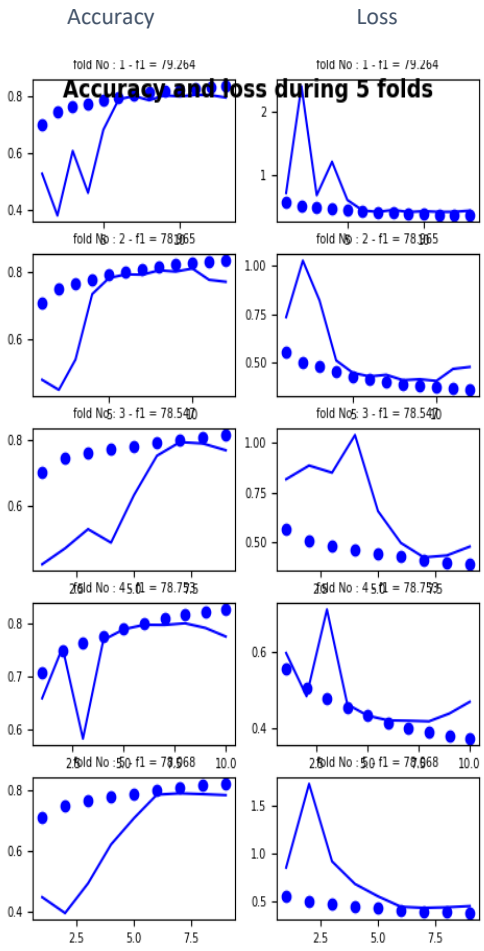


Figure 24

[Method -2]

x-axis : epochs , y- axis : accuracy

dotted line for training set

filled line for validation set that uses keras in order to stop training

Models Architecture

The full architecture of our models can be seen at the following figures (Figure 25, 26)

Method-1

Siamese deepNet - GloVe 300D

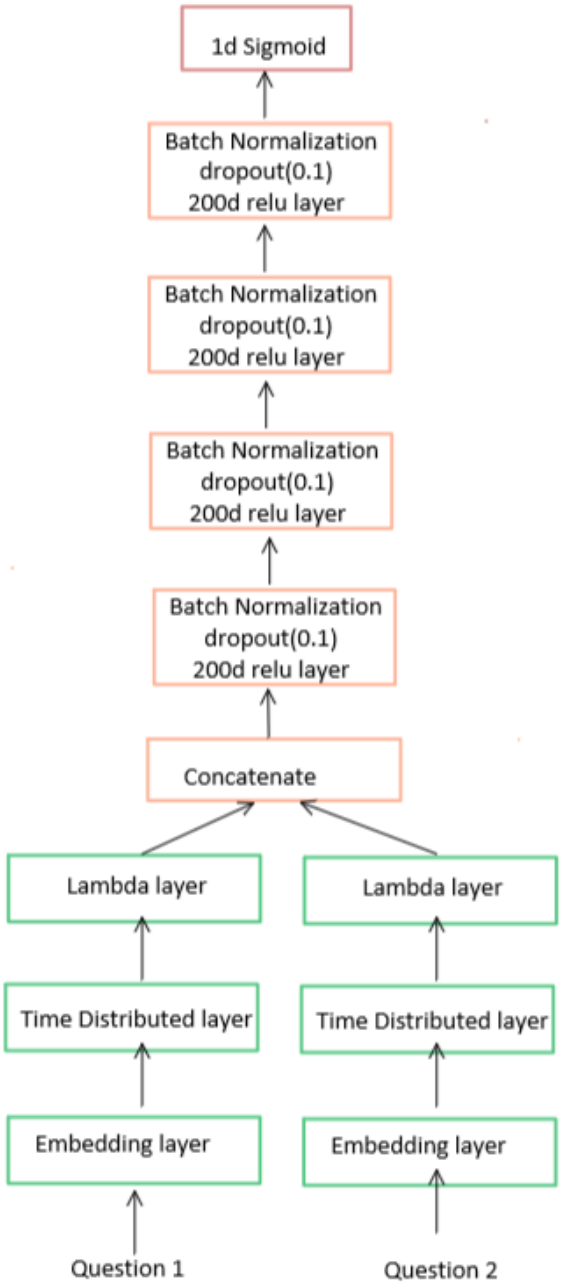


Figure 25

Method-2

Siamese with distance metrics - Word2Vec

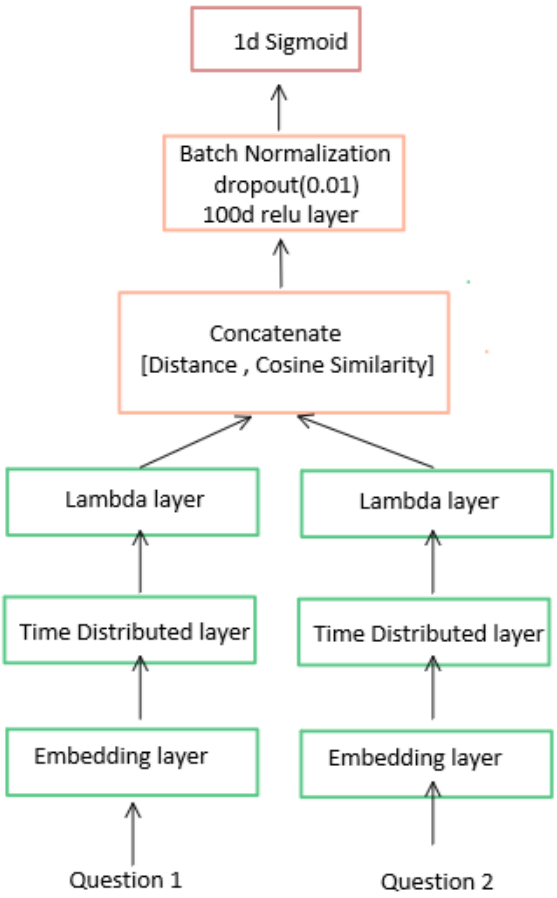


Figure 26

TimeDistributed Layers (Figure 27) are used for temporal data when we want to apply the same transformation to every time-step. In our data set, each question has 25 words which correspond to 25 time steps. We use a dense layer with 300 hidden inputs — since our data has 300 dimensional embedding, we get 90,000 + 300 (bias) = 90,300 weights for the layer. Both question 1 and question 2 pass through similar time distributed layers.

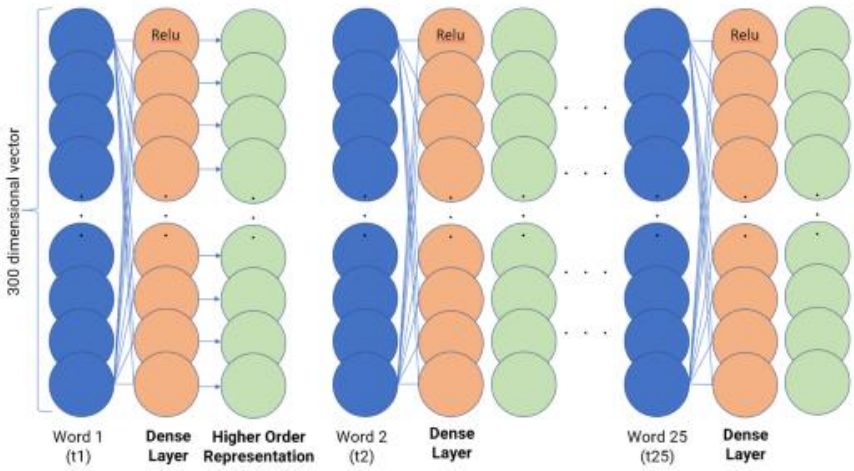


Figure 27

Each of the 300 hidden units in the time distributed dense layer (shown in orange) connect with the word vectors at each time step (shown in blue) and produce higher order representations (shown in green). All the dense layer units have the 'Relu' activation for non-linearity.

Lambda Layers (Figure 28) Keras allows us to use custom layers in our model. Lambda layer is often used on the higher order representations obtained after the time distributed dense layers to encapsulate the meaning of the meanings of all the words in the question i .e of the entire question in 300 dimensions.

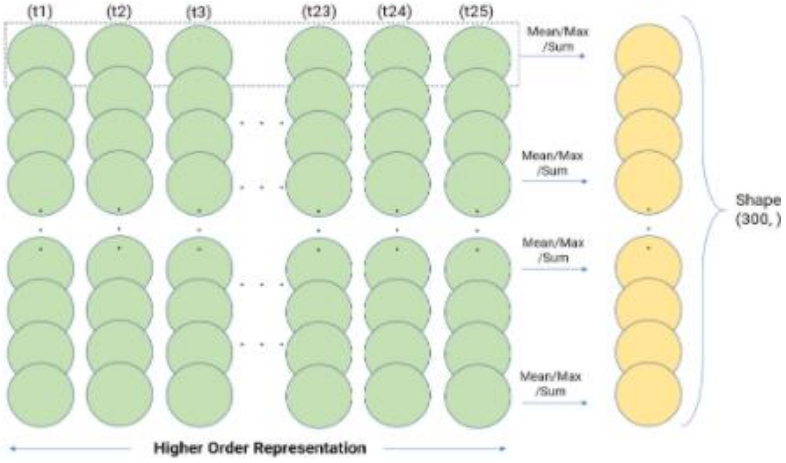


Figure 28

For that reason, compute an aggregate representation by calculating the average, max, sum etc. For Method -1 we used max, when for Method - 2 we used sum.

Dense Layer Fully Connected Layer , Activation function relu

Dropout Layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Dropout follows the assumption that different feature combinations can represent different aspects of the scene that we wish to describe. Thus, at each training stage,individual nodes are either ”dropped out” of the network with probability 1-p or kept with probability p so that a reduced network is left. Only the reduced network is trained on the data in that stage and the removed nodes are reinserted into the network in the next stage with their original weights.

Batch Normalization Layer applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

Output Layer, Activation Function : Sigmoid

> *When a prediction took part in, If output is less than 0.5 the predicted classes were assigned with “0” label else with label “1”. (See **make_prediction**(model, X_test))*

To state an example of question pairs with the output of dense layer and the predicted class are listed in the table below:

ID	Question 1	Question 2	Output of Siamese deepNet	Predicted Label (isDuplicate)
283003	What can someone do if they've lost the wireless USB connector to their Logitech keyboard and mouse?	What is the best USB wireless mouse that can be used for both a PC and a Mac?	0.518337	1
283004	Why India need to elect Prime minister?	Is prime minister of India elected or appointed?	0.004974	0
283005	How can I make money online with free of cost?	How can I make money online for free?	0.204776	0
283006	Does MDMA affect the first and higher order moments of neuron firing rates (like mean/variance/skew)? If so, how?	Do antipsychotics affect the first and higher order moments of neuron firing rates (like mean/variance/skew)? If so, how?	0.969477	1

Lastly,

- Method 1 simply concatenates the outputs of the Siamese.
- Method 2 after Siamese building computes the Squared Euclidean Distance and Cosine Similarity of two outputs. Then the two metrics are concatenated and fit the last dense layer .

```
def Angle(inputs):

    length_input_1=K.sqrt(K.sum(tf.pow(inputs[0],2),axis=1,keepdims=True))
```

```
    length_input_2=K.sqrt(K.sum(tf.pow(inputs[1],2),axis=1,keepdims=True))
    result=K.batch_dot(inputs[0],inputs[1],axes=1)/(length_input_1*length_input_2)
    angle = tf.acos(result)
    return angle
def Distance(inputs):
    s = inputs[0] - inputs[1]
    print(s)
    output = K.sum(s ** 2,axis=1,keepdims=True)
    return output
```

- Every method uses “adam” Optimizer
- Every method uses early stop which monitors the validation accuracy since performance of the model is evaluated on the validation set at the end of each epoch. Validation set is defined at the end of the first epoch using validation_split. Helps avoid overfitting.

Full code for Method-1 can be found in 2b_siamese_deepNet.py

Full code for Method-2 can be found in 2b_siamese_with_distances.py

Results

Methods	**Precision**	**Recall**	**F-Measure**	**Accuracy**
Method-1: Siamese deepNet - GloVe 300D	77.40	77.42	77.40	78.89
Method-2: Siamese with distance metrics - Word2Vec	78.50	79.26	78.71	79.81

As we can observe, two methods had almost equal performance. Method-2 had the better results on Kaggle prediction: 81.033 % (Method-1 had scored 80.521%).

Suggestions for beating the benchmark

Data augmentation could be a good technique in order to multiply our training set and reduce class imbalance. Over fitting will also be reduced. Homma et al. (2016) reached almost 99% using data augmentation.

The file with the training data includes some errors, there are cases where label 1 had to be label 0, or the opposite, so if someone fixes the labels better accuracy can be achieved .

References

Bogdanova, Santos, C., Barbosa , L., D.,Zadrozny, B., (2015) Learning Hybrid Representations to Retrieve Semantically Equivalent Questions. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers), 694–699, https://www.aclweb.org/anthology/P15-2114/

Bowman, S., Angeli, G., Potts , C., Manning, C.(2015) A large annotated corpus for learning natural language inference. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, 632–642, https://www.aclweb.org/anthology/D15-1075/

Brownlee J. (2021, February 8), Singular Value Decomposition for Dimensionality Reduction in Python. https://machinelearningmastery.com/singular-value-decomposition-for-dimensionality-reduction-in-python/

Chollet, F., (2017). Deep Learning with Python (1st. ed.). Manning Publications Co., USA.

Bryłkowski, H. (2021, February 8), Locality sensitive hashing — LSH explained. https://medium.com/engineering-brainly/locality-sensitive-hashing-explained-304eb39291e4

cmhteixeira (2021, February 8), Locality Sensitive Hashing (LSH). https://aerodatablog.wordpress.com/2017/11/29/locality-sensitive-hashing-lsh/

ekzhu (2021, February 8), http://ekzhu.com/datasketch/lsh.html

Eledath, B. (2021, February 8), Finding Duplicate Questions using DataSketch. https://medium.com/@bassimfaizal/finding-duplicate-questions-using-datasketch-2ae1f3d8bc5c

Gupta S. (2021, February 8), Locality Sensitive Hashing: An effective way of reducing the dimensionality of your data. https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134

Leskovec, J., Rajaraman, A., Ullman., J. (2014) Mining of Massive Datasets (2nd. ed.). Cambridge University Press, USA. http://www.mmds.org/

MLNerds (2021, February 8), What is the difference between word2Vec and Glove? https://machinelearninginterview.com/topics/natural-language-processing/what-is-the-difference-between-word2vec-and-glove/

Pande K. (2021, February 8), Question pairs identification: There are no stupid questions…only duplicates! https://towardsdatascience.com/questions-pairs-identification-f8abcafb5b17

Ross, E. (2021, February 8), .Searching for Near Duplicates with Minhash. https://skeptric.com/minhash-lsh/

Santhosh H. (2021, February 8), Locality Sensitive Hashing for Similar Item Search. https://towardsdatascience.com/locality-sensitive-hashing-for-music-search-f2f1940ace23

(2021, February 8), https://keras.io/