Project Report

Requirement 1: Text classification

In this part of the assignment it was required to perform text classification given a train test set with news articles. We applied vectorization techniques to represent text data and applied such machine learning classification models as Random Forests, Support Vector Machines (SVM) and K-Nearest Neighbour (KNN) to predict the class of the given article in a test set.

Question 1a: Get to know the Data: WordCloud

To get familiar with data and obtain the general description of each class, or category, word clouds were created using the Python WordCloud library.

To accomplish this task:

1. The data frame for the train set was split into four parts, where each part corresponds to one of the given categories: Business, Entertainment, Health, Technology.
2. As it was mentioned earlier, the WordCloud library was chosen with the STOPWORDS parameter enabled. STOPWORDS parameter from this library provides 190 stopwords whereas another famous tool nltk.corpus (stopwords 'English') only 178, therefore we used the first one.
3. Afterwards word cloud images were generated using matplotlib.pyplot feature plt.
4. In the process of the generation WordCloud images were reviewed and additional stopwords were added ("said", "say", "may", "says", "one", "even", "now", "well", "will") to improve WordCloud image accuracy for representing each category.
5. Finally regenerated WordCloud images were evaluated on their compliance to each category. The figures for each category are shown below.

Generating word cloud images was useful to identify the words common for each category and remove them. This in turn will help to improve the classification of the data, which is the task of the next section.

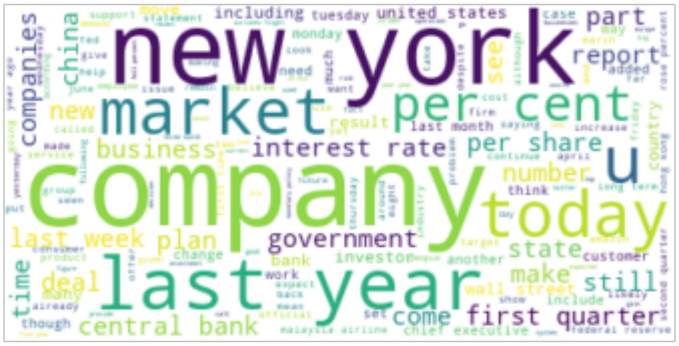


Figure 1. Word cloud image for Business category.

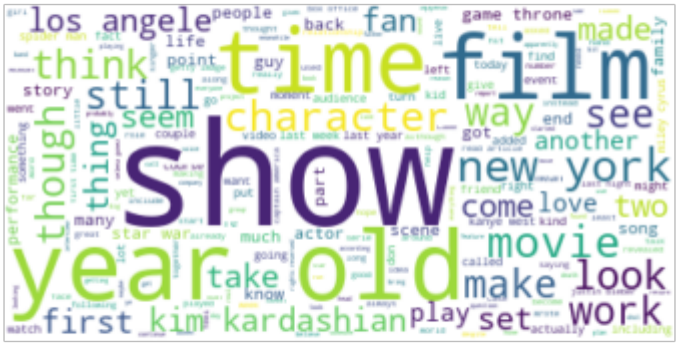


Figure 2. Word cloud image for Entertainment category.

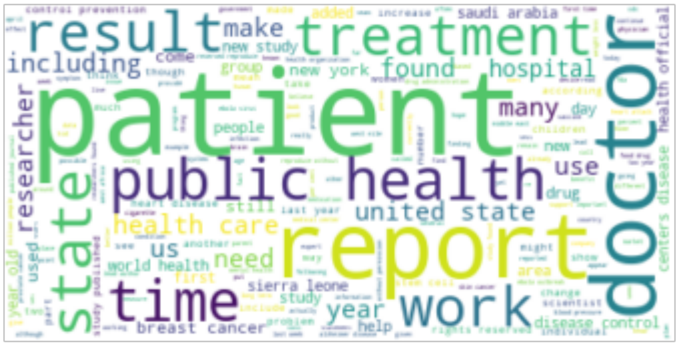


Figure 3. Word cloud image for Health category.

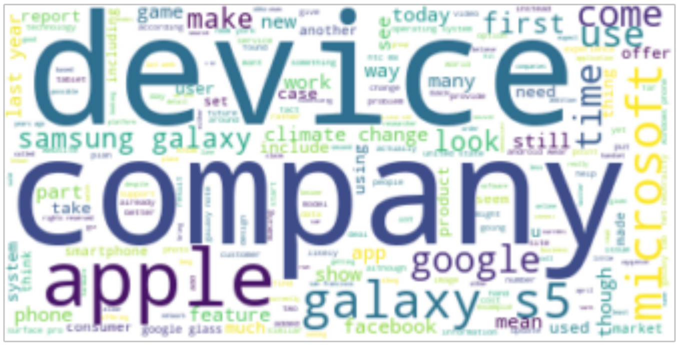


Figure 4. Word cloud image for technology category.

Question 1b: Classification Task

In this task we performed actual classification of the articles using Random Forests (RF) and Support Vector Machines (SVM).

Data preparation

Bag-of-Words (BoW)

Text classification is considered as a sentiment analysis problem, and one of the techniques used to build such sentiment analysis models is a bag-of-words (BoW) [14]. The main idea is to represent each word in a document as a key with a value equal to the number of occurrences of that word in a given document. If the analyzing document is large, the BoW model will result in huge sparse matrices, where many values will be 0.

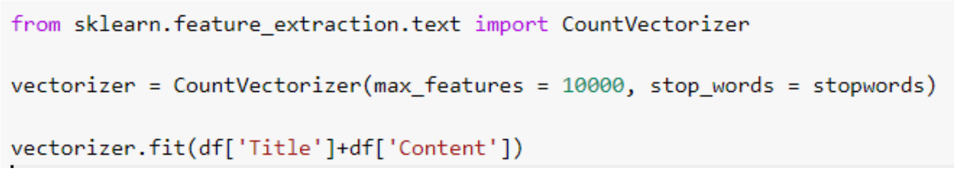
To apply the BoW method we used vectorizers available in the Scikit-learn library. They accomplish the following tasks:

- Split the text samples into word n-grams
- Data vectorization (convert n-grams into vectors)

We used sklearn.feature_extraction.text library which includes CountVectorizer.

CountVectorizer was configured to process data:

- extract limit to 10000 top features (max_features=1000),
- STOPWORDS parameter was set to the stopwords list identified when creating the WordCloud images.

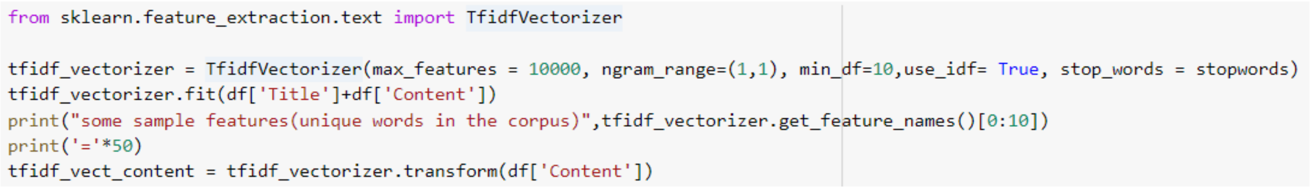


```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(max_features = 10000, stop_words = stopwords)

vectorizer.fit(df['Title']+df['Content'])
```

Also for comparison between data preprocessing quality TfidfVectorizer was applied from the same library. TfidfVectorizer uses 3 main data preprocessing steps: compute word counts, computes Inverse Document Frequency (IDF) and Tf-idf scores.

TfidfVectorizer was configured to process data:

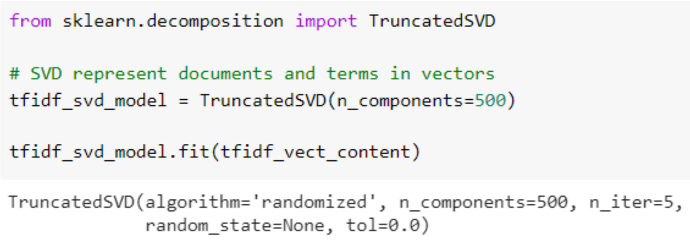
- extract limit to 10000 top features (max_features=1000),
- range of n-gram sizes for tokenizing text (1,1),
- minimum document/corpus frequency below which a token is discarded is 10 (min_df=10),
- STOPWORDS parameter was set to the stopwords list identified when creating the WordCloud images.



```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(max_features = 10000, ngram_range=(1,1), min_df=10,use_idf= True, stop_words = stopwords)
tfidf_vectorizer.fit(df['Title']+df['Content'])
print("some sample features(unique words in the corpus)",tfidf_vectorizer.get_feature_names()[0:10])
print('='*50)
tfidf_vect_content = tfidf_vectorizer.transform(df['Content'])
```

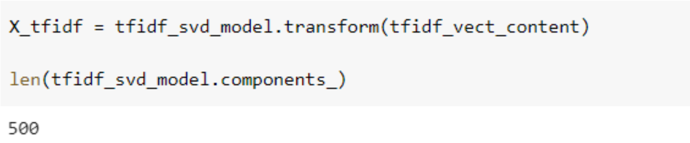
Bag-of-Words (BoW) + Singular Value Decomposition (SVD)

Since the resultant data matrix created using the BoW model has a very high dimension to achieve better computation efficiency, the Singular Value Decomposition (SVD) method was applied. It is a common technique applied for dimensionality reduction, image compression and noise reduction [15]. The code snippet below shows how the Truncated SVD method was from sklearn.decomposition library to transform the vectorized content and reduce the number of features to 500.



```
from sklearn.decomposition import TruncatedSVD

# SVD represent documents and terms in vectors
tfidf_svd_model = TruncatedSVD(n_components=500)

tfidf_svd_model.fit(tfidf_vect_content)
```

```
TruncatedSVD(algorithm='randomized', n_components=500, n_iter=5,
             random_state=None, tol=0.0)
```



```
X_tfidf = tfidf_svd_model.transform(tfidf_vect_content)

len(tfidf_svd_model.components_)
```

```
500
```

Model building and training

The obtained vectorized data from both TfidfVectorizer and CountVectorizer were used in 3 different algorithms. We used implementations of Random Forest and SVM algorithms from the Scikit-learn library. LinearSVC() implementation was chosen due to lower time complexity. Our third algorithm of choice was K-Nearest Neighbour (KNN) with the number of neighbors equal to 5 (also from the Scikit-learn library).

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**	**SVM (SVD)**	**Random Forest (SVD)**	**(KNN) (BoW)**	**(KNN) (SVD)**
Accuracy	0.9465	0.9379	0.9324	0.9207	0.902	0.9326
Precision	0.9421	0.9382	0.9299	0.9238	0.9043	0.9281
Recall	0.9406	0.9266	0.9214	0.8989	0.892	0.9247
F-Measure	0.9413	0.9321	0.9254	0.9100	0.8964	0.9261

Table 1. Evaluation results using the train set and 5-fold cross-validation (with Count Vectorization)

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**	**SVM (SVD)**	**Random Forest (SVD)**	**(KNN) (BoW)**	**(KNN) (SVD)**
Accuracy	0.9654	0.9369	0.9455	0.9519	0.9693	0.9633
Precision	0.9634	0.9371	0.942	0.9508	0.9668	0.963
Recall	0.9611	0.9248	0.9371	0.9416	0.9653	0.9578
F-Measure	0.9622	0.9306	0.9395	0.946	0.966	0.9603

Table 2. Evaluation results using the train set and 5-fold cross-validation (with TfIdf Vectorization)

Tables 1 and Table 2 show the results for 5-fold cross-validation with both types of Vectorizers. KNN classifier with BoW method using TfidfVectorizer (see Table 2) performs the best. The model was used to get labels in a test data and output file “testSet_categories.csv”. After uploading the output .csv file to the Kaggle platform this model achieved an accuracy score of 0.9692.

Beat the Benchmark

As an experiment to achieve better validation accuracy and also improve model accuracy, F-Measure, Precision, Recall metrics we tried to change TfidfVectorizer parameters (see BigData_Project_Question1_KNN(5)+NewVectorizer.ipynb file) :

- extract limit to 20000 top features (max_features=20000),
- range of n-gram sizes for tokenizing text (1,2),
- minimum document/corpus frequency below which a token is discarded is 2 (min_df=2),

Since from the previous experiments we found out that KNN is the best classifier we rerun only this model with a new vectorizer. The results are presented in Table 3.

Statistic Measure	**(KNN)** **(BoW)** Max_features =20000	**(KNN)** **(SVD)** Max_features =20000	**(KNN)** **(BoW)**	**(KNN)** **(SVD)**
Accuracy	0.9695	0.9632	0.9693	0.9633
Precision	0.9667	0.9621	0.9668	0.963
Recall	0.9658	0.9571	0.9653	0.9578
F-Measure	0.96623	0.9596	0.966	0.9603

Table 3. Evaluation results using the train set and 5-fold cross-validation for KNN model (with modified TfIdf Vectorization)

With the unlabeled test set in Kaggle, the final accuracy has slightly increased and achieved 0.9708. The test accuracy resulted to be higher than the validation obtained with 5-fold cross-validation. However, the result from the Kaggle platform might change since 10% of the test dataset was left for final scores calculation.

All in all, it can be concluded that we were able to beat our best model by experimenting with the preprocessing steps such as using different types of vectorizers and tuning the best vectorizer parameters. KNN model using TfidfVectorizer the modified parameters showed the best performance.

Requirement 2: Nearest Neighbor Search and Duplicate Detection

Question 2α: De-Duplication with Locality Sensitive Hashing

The aim of this task was to detect the number of duplicated documents given two sets of the documents, i.e. to identify how many similar documents in a test set are present in a train set. Each document is a Quora question, therefore the task can be referred to as the similarity score calculation between short sentences. The similarity threshold was set to 0.8.

Similarity scores were calculated using four different methods:

1. Exact-Cosine method
2. Exact-Jaccard method
3. Cosine Similarity: Random projection LSH family.
4. Jaccard Similarity: Min-Hash LSH family

It should be noted that the similarity detection based on the above methods belongs to a Term-Based Similarity Measure technique, hence these methods alone can not handle perfectly the semantic meaning of the textual information [1].

Next, each of the methods will be briefly discussed.

Method 1. Exact-Cosine De-Duplication

Cosine similarity is a widely used technique to derive the similarity between two sentences by modeling text as term vectors and calculating the cosine value between those two vectors [2].

The similarity score can be computed using the following formula:

$$Sim(\vec{q},\vec{d}) = \frac{\vec{q}.\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_{k=1}^{t} w_{qk} \times w_{dk}}{\sqrt{\sum_{k=1}^{t}(w_{qk})^2} . \sqrt{\sum_{k=1}^{t}(w_{dk})^2}}$$

where d and q are document (pool of the questions) and query vectors respectively. The terms in the right side of the equation represent float numbers indicate the frequency of each instance inside the document.

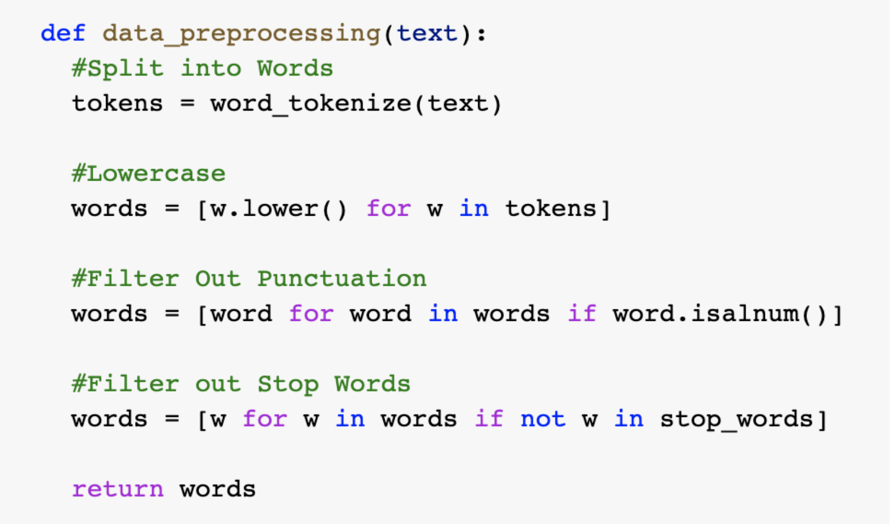
We used the TfIdf Vectorizer to obtain the term vectors and cosine similarity implementations from the Scikit-learn library to calculate the final score. The number of duplicates was calculated based on the following condition: if a cosine similarity score of a single row in a test set exceeds 0.8 when compared to a sparse matrix built from the train content, then it is considered as a duplicate.

Method 2. Exact-Jaccard De-Duplication

Jaccard Similarity is another method for determining the similarity between two sentences [3]. Originally, it is a measure of similarity between two sets and can be calculated as follows:

$$J(S,T) = \frac{|S \cap T|}{|S \cup T|}$$

Therefore first we converted all the questions into a set of words and applied some data cleaning such as lowercasing, eliminating special characters except alphanumeric ones and removing common stopwords in English defined by Python Natural Language Tooklit. The steps of the algorithm are shown in a code snippet in Figure 5.



```
def data_preprocessing(text):
  #Split into Words
  tokens = word_tokenize(text)

  #Lowercase
  words = [w.lower() for w in tokens]

  #Filter Out Punctuation
  words = [word for word in words if word.isalnum()]

  #Filter out Stop Words
  words = [w for w in words if not w in stop_words]

  return words
```

Figure 5. Data Preprocessing function.

After cleaning data we calculated Jaccard similarity between each pair of questions in train and tests, and following the same rule as with the Exact-Cosine method found the number of the duplicates. Since we compared each sentence in a test content with each sentence in a train content, this procedure was not time efficient. In the Exact-Cosine method the power of vectorized operations was leveraged to avoid elementwise comparison, which was not the case in Exact-

Jaccard. Therefore in the next methods of using Locality Sensitive Hashing this issue was addressed.

Method 3. LSH-Cosine De-Duplication

The main idea behind Locality Sensitive Hashing (LSH) is to map the given points that are close to each other to the same bucket to decrease the search space for a given query [11]. This method is more computationally efficient than exact calculations for both Cosine and Jaccard.

In the case of LSH-cosine we create random projections (vectors defining the hyperplane) to separate the points in subspaces, each subspace representing a bucket [12]. During the training of the vectorized content of our train data we obtained bin indices for each row (by multiplying them with the generated random vectors) and encoded bis as integers. Each bit index represents one bucket in a created hash table (using Python dictionary). After organizing the whole train content in a hash table we saved the model with all necessary parameters including the set of random vectors. Those random vectors are used again to generate the bin index for a given query test vector. When a matching bin index is found we again used the cosine_similarity function to obtain the scores and selected only those items which satisfy the given threshold. The whole procedure was repeated with different values of a parameter K, from 1 to 10, that defines the number of generated hyperplanes.

Method 4. LSH-Jaccard De-Duplication

LSH-Jaccard also provides an advantage of computational efficiency, but it is based on calculating minHash values based on Jaccard Similarity via a representation of the text as a set of words [4]. Similarly, it organizes the train data into a hash table that can be queried later.

As it was suggested we used a datasketch library to implement this task. After inserting each element of the preprocessed data (the same preprocessing function again) to the hash table (MinHashLSH) we queried all the questions in test data and those questions that satisfy the threshold condition were returned.

Results and discussion

Table 4. Evaluation results for all methods of De-Duplication

Type	Build Time	Query Time	Total Time	#Duplicates	Parameters	Permutations
Exact-Cosine	-	333	333	1908	-	-
Exact-Jaccard	-	5321	5321	2191	-	-

LSH-Cosine	880	196	996	1630	1	-
LSH-Cosine	460	92	552	1552	2	-
LSH-Cosine	289	42	331	1387	3	-
LSH-Cosine	197	22	219	1211	4	-
LSH-Cosine	142	13	155	888	5	-
LSH-Cosine	111	8	119	944	6	-
LSH-Cosine	101	6	107	832	7	-
LSH-Cosine	99	5	104	720	8	-
LSH-Cosine	99	4	103	679	9	-
LSH-Cosine	98	4	102	656	10	-
LSH-Jaccard	423	4	427	8547	-	16
LSH-Jaccard	490	5	495	7778	-	32
LSH-Jaccard	625	6	631	7910	-	64

Table 4 shows the experimental results for each of the above applied methods. As it is evident Exact methods, especially Exact-Jaccard, take a considerable amount of time to answer all the question queries in test data. The number of duplicates identified by both methods was always the same.

With Locality Sensitive Hashing we were able to significantly improve the look-up time (Query time) for both Cosine and Jaccard similarity measures, though building the models took some time. For LSH-Jaccard with the increasing number of permutations higher number of duplicate questions were detected at the expense of both build and query time. In contrast, with the increasing value of the parameter K, the general trend was the drop of the build and query time, but fewer duplicates were found.

In general, the LSH-Jaccard method identified a significantly higher number of duplicates compared to other methods. This can be attributed to the fact that this method is not exact and subject to a huge number of false positives and false negatives [4]. False positives appear when the items in the same entry although their similarity is below the specified threshold. Alternatively, false negatives are those items that were mapped to the separate entries although the similarity satisfies the threshold conditions.

The preference of one method over another depends on the settings and the specific requirements of the similarity detection task. For identifying the number of similar questions to a target one for a large pool of questions, it is better to use Locality Sensitive Hashing, which ensures quick response since the query time is small as proven by our results.

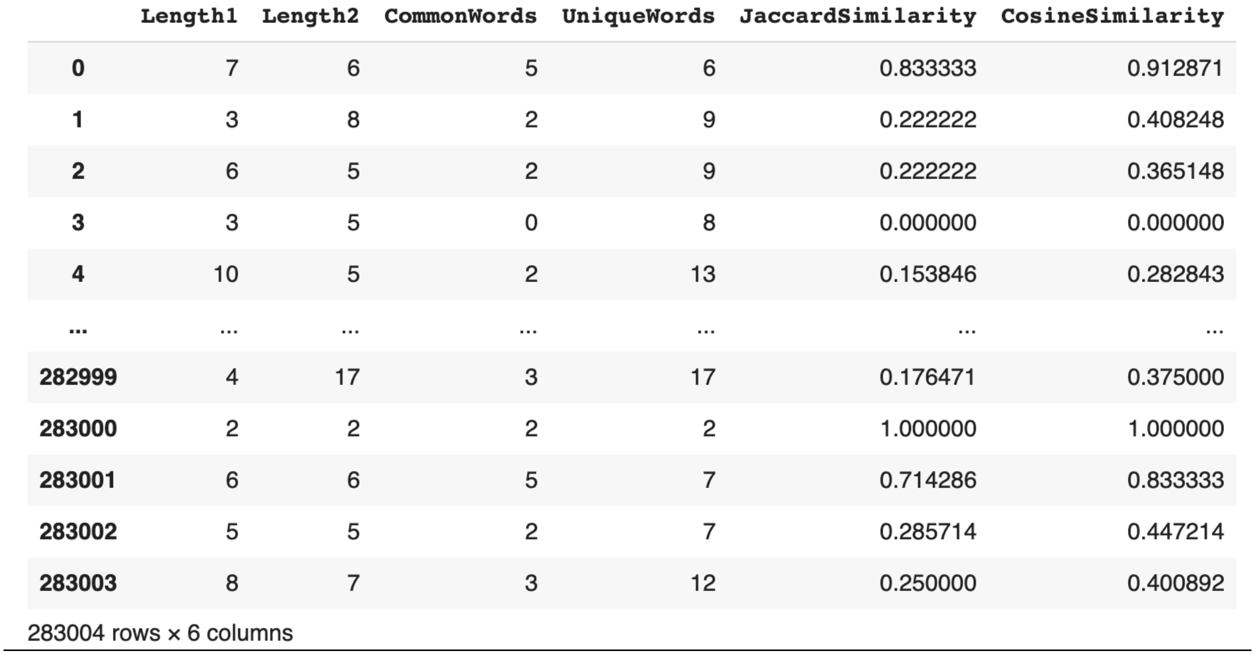
Question 2b: Same Question Detection

In this part of the assignment we implemented the model which given a pair of two questions decides if they are identical or not. This task also can be considered as a sentence similarity problem however here our similarity evaluation should be based on a contextual meaning of the sentence rather than solely on a number of shared similar words. In other words, the semantic similarity between two sentences should be measured by recognizing the semantic relations between them [1].

Again, train and test datasets were given, containing a huge number of the Quora questions. Both datasets contain such information as a unique ID for each pair of the questions, the first question and the second question. The train set was already labeled for each pair, and the final model was supposed to be able to predict the labels of the test set. The goal was to experiment with various heuristic features related to the sentence similarity, train the model with the train set and output final predictions.

Two methods were used to complete this task. The first one is based on deriving simple features describing two sentences such as their length, the number of common and unique items for each pair. Also, Jaccard and Cosine similarity scores were computed. Although these features do not infer alone the semantic meaning of the sentences, the data-driven approach that we apply here makes it possible to distinguish sentences based on the learned patterns. The second method is an extension of the initial model. We tried to apply the method of semantic similarity measurement and add it as an additional feature to the model.

Method 1. Simple approach based on syntactic feature extraction



	Length1	Length2	CommonWords	UniqueWords	JaccardSimilarity	CosineSimilarity
0	7	6	5	6	0.833333	0.912871
1	3	8	2	9	0.222222	0.408248
2	6	5	2	9	0.222222	0.365148
3	3	5	0	8	0.000000	0.000000
4	10	5	2	13	0.153846	0.282843
...	...	...	...	...	...	...
282999	4	17	3	17	0.176471	0.375000
283000	2	2	2	2	1.000000	1.000000
283001	6	6	5	7	0.714286	0.833333
283002	5	5	2	7	0.285714	0.447214
283003	8	7	3	12	0.250000	0.400892

283004 rows × 6 columns

Figure 6. A new dataframe created from the extracted features.

The method has a number of steps:

1. First train and test sets were checked for the presence of any missing, or Null, values. Two missing values were found for the train set and one - for the test set. Instead of dropping the rows containing the missing values we decided to replace them with empty strings to keep the original data sizes.
2. Checking for duplicate rows also was done and none were identified.
3. Data preprocessing was applied using the same function defined in Figure 5. We avoided using stemming or lemmatization techniques, because they could potentially destroy the word meaning thereby worsening the model's accuracy.
4. Length1 and Length2 features were created by creating the lists containing the lengths of all rows in Question1 and Question2 columns.
5. For each pair of questions the lists containing the number of summon words (intersection) and unique (union) were calculated and stored as features CommonWords and UniqueWords.
6. JaccardSimilarity and CosineSimilarity features were extracted by applying these algorithms defined in a previous section. Here instead of applying TfIdf vectorization simple term vectors were built based on the occurrence of the words in a union set.
7. Using a new Pandas Dataframe (see Figure 6) constructed from all the features extracted we trained the machine learning model with the Random Forest classifier using k-fold (k = 5) cross-validation (both from Scikit-learn library). It should be noted that single-fold validation was tried first with other models such as Linear SVM and KNN, however,

since Random Forest resulted in a higher accuracy it was selected as the final classification model.

8. Finally, the model was trained using the whole dataset. After applying similar data preparation procedures described in steps 1-6, the predictions were obtained for the test dataset and saved in an output .csv file as instructed.

Method 2. Combination with the Semantic Similarity

Finding the semantic sentence similarity between two sentences is a non-trivial task. As it was mentioned before while the syntactic approach implemented in Method 1 considers only co-occurring words in a given sentence, the semantic approach attempts to calculate the similarity score between words based on a Semantic Net [6].

The algorithm that we used for this part is well described in [5]. The following extract from the paper well defines the problem:

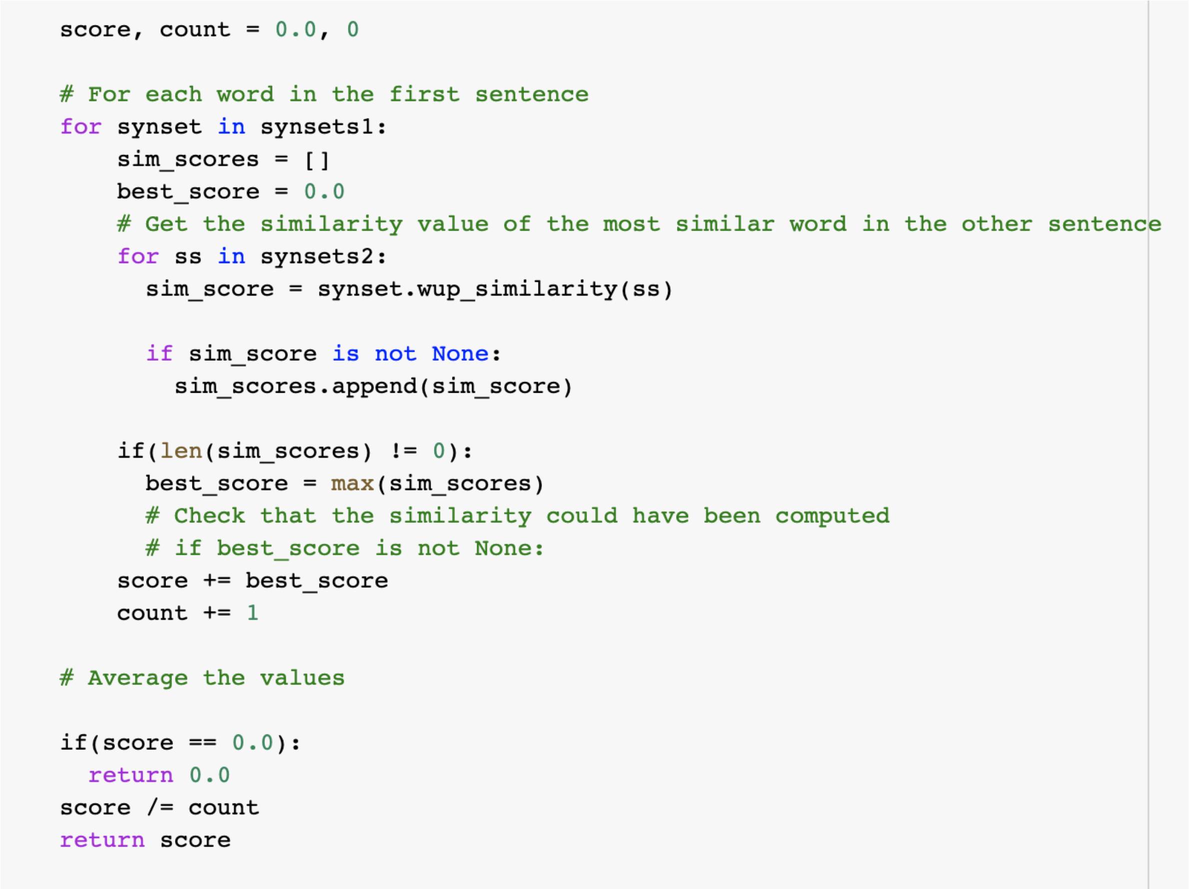
Given a metric for word-to-word similarity and a measure of word specificity, we define the semantic similarity of two text segments T1 and T2 using a metric that combines the semantic similarities of each text segment in turn with respect to the other text segment.

We slightly modified the implementation of this algorithm by [7] to fit it to our dataset. Also, we used Wu and Palmer similarity metric measure, which derives similarity score as follows:

$$Sim_{wup} = \frac{2 * depth(LCS)}{depth(concept_1) + depth(concept_2)}$$

,

where depth(concept) is the depth of a given word in WordNet taxonomy and depth (LCS) is the depth of the least common subsumer. WordNet is a lexical database that contains a huge number of nouns, verbs, adjectives and adverbs organized in a set of synonyms (synsets) [6]. We used the wordnet library from Scikit-learn.

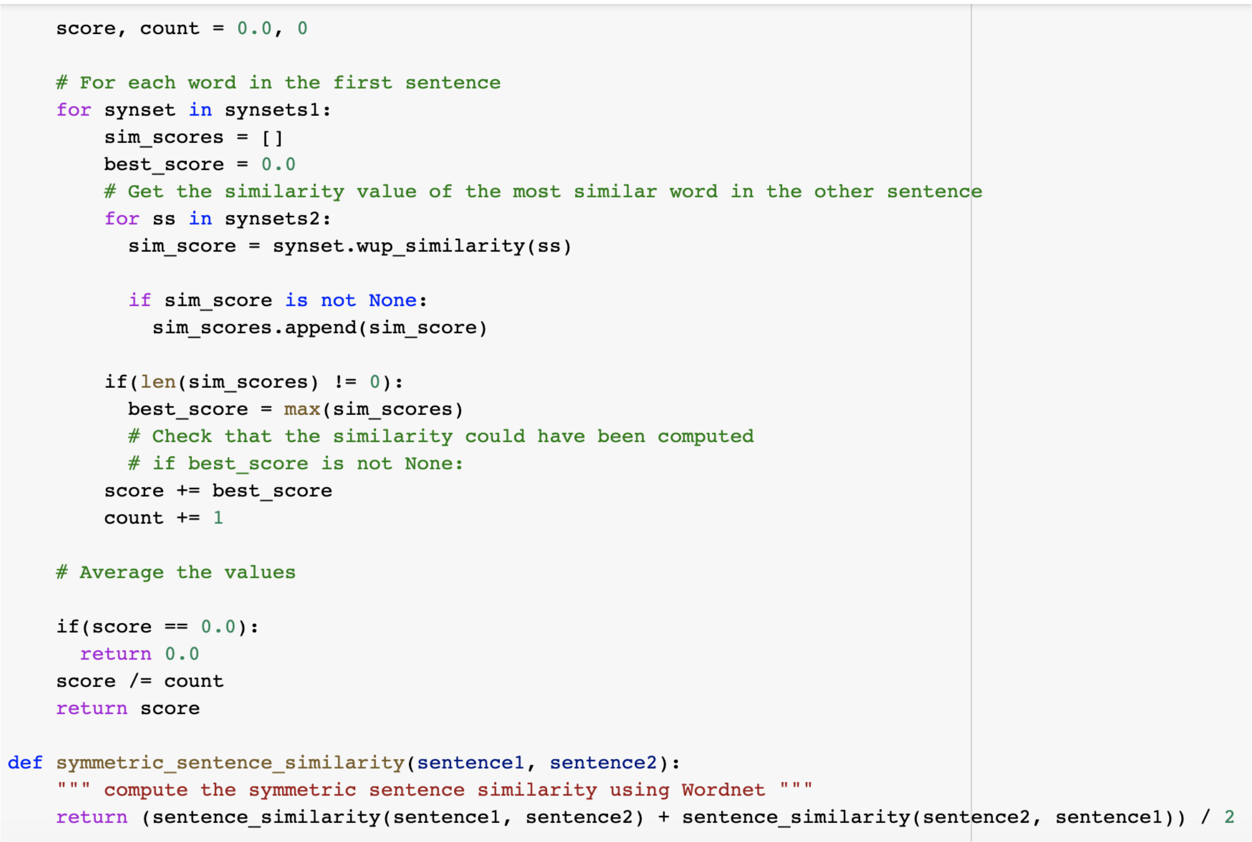


```
score, count = 0.0, 0

# For each word in the first sentence
for synset in synsets1:
    sim_scores = []
    best_score = 0.0
    # Get the similarity value of the most similar word in the other sentence
    for ss in synsets2:
      sim_score = synset.wup_similarity(ss)

      if sim_score is not None:
        sim_scores.append(sim_score)

    if(len(sim_scores) != 0):
      best_score = max(sim_scores)
      # Check that the similarity could have been computed
      # if best_score is not None:
    score += best_score
    count += 1

# Average the values

if(score == 0.0):
  return 0.0
score /= count
return score
```

Figure 7. Preparing data for WordNet.

Our method can be described as following:

1. First, we POS (Part Of Speech) tag our preprocessed sentences to know to which part of sentences they belong.
2. Then after converting tags to the form that can be understood by WordNet we obtained the synsets for each word in a sentence. These two steps are shown in a code snippet in Figure 7.



```
    score, count = 0.0, 0

    # For each word in the first sentence
    for synset in synsets1:
        sim_scores = []
        best_score = 0.0
        # Get the similarity value of the most similar word in the other sentence
        for ss in synsets2:
          sim_score = synset.wup_similarity(ss)

          if sim_score is not None:
            sim_scores.append(sim_score)

        if(len(sim_scores) != 0):
          best_score = max(sim_scores)
          # Check that the similarity could have been computed
          # if best_score is not None:
        score += best_score
        count += 1

    # Average the values

    if(score == 0.0):
      return 0.0
    score /= count
    return score

def symmetric_sentence_similarity(sentence1, sentence2):
    """ compute the symmetric sentence similarity using Wordnet """
    return (sentence_similarity(sentence1, sentence2) + sentence_similarity(sentence2, sentence1)) / 2
```

Figure 8. Semantic Similarity calculation.

3. Then for loop shown in Figure 8 performs the main similarity calculation by finding the best score via wup_similarity for each word in sentence 1 against each word in sentence 2. The final score is normalized by the length of the first sentence. This procedure is repeated for the second sentence provided by the function symmetric_sentence_similarity and the average score is returned.

All in all, in Method 2 similarly to the authors of work [6] we tried to build a hybrid combining both syntactic and semantic approaches. Hence, in a similar manner as obtained Jaccard and Cosine Similarity scores for each question pair we calculated semantic similarity score as an additional feature to put in a new train dataframe. Then steps 7-8 were from Method 1 were repeated with an enhanced feature dataset and the final predictions were derived.

Results and discussion

Table 5. Evaluation results for the methods of Same Question Detection

Method	**Precision**	**Recall**	**F-Measure**	**Accuracy**
Method-1	0.6818	0.6854	0.6833	0.7011
Method-2	0.6700	0.6743	0.6717	0.6893

Table 5 represents the results for the methods applied for Same Question Detection. With Method 1 a slightly higher prediction accuracy was achieved, than with Method 2, 0.70 against 0.69. Therefore we decided to choose the second model to make the final predictions for the test set without labels.

The second method's inability to improve the model can be explained by several drawbacks of the selected algorithm. First, the WordNet database is not able to find the synonyms for some given words, so because of that, the similarity score for those words was set to 0. Besides, the algorithm is not identical to the one proposed in [5] because we did not take into account inverse-document-frequency. The order of the words also was not considered although it can affect the meaning of the sentence.

We achieved an accuracy of 0.75 on a labeled test set checked in Kaggle. Even though 90% of the test data was checked, 5% enhancement of the test accuracy over validation accuracy is a rare case in machine learning practice. This can be explained by the fact that data annotation for sentence similarity is a subjective task and thus can lead to some instances labeled incorrectly in a test set, while the algorithm made better predictions for them. Due to similar reasons for the wrong question pairs annotation for the train set, the model could not achieve very high accuracies.

It should be noted that after literature review it was found that the semantic sentence similarity task is dependent on the given dataset. According to [8], where authors analyzed various existing techniques only for the datasets with a high degree of the overlap reasonable accuracies were achieved, whereas in the majority of cases they did not exceed 70%. The authors also used here WordNet database and mentioned its limited coverage of words.

There are a lot of proposed methods in literature to compute the semantic sentence similarity, and experimenting with them might improve the results for this task. One promising approach that was mentioned in a recent study [9] is an application of deep learning that has the potential to improve the accuracy of the sentence similarity task by capturing the key features of the sentences.

References

[1] Majumder, Goutam & Pakray, Dr. Partha & Gelbukh, Alexander & Pinto, David. (2016). Semantic Textual Similarity Methods, Tools, and Applications: A Survey. Computacion y Sistemas. 20. 647-665. 10.13053/CyS-20-4-2506.

[2] Rahutomo, Faisal & Kitasuka, Teruaki & Aritsugi, Masayoshi. (2012). Semantic Cosine Similarity.

[3] Fletcher, Sam & Islam, Md. (2018). Comparing sets of patterns with the Jaccard index. Australasian Journal of Information Systems. 22. 10.3127/ajis.v22i0.1538.

[4] C., & Cmhteixeira, V. A. P. B. (2019, March 10). *Locality Sensitive Hashing (LSH)*. Aerodata. https://aerodatablog.wordpress.com/2017/11/29/locality-sensitive-hashing-lsh/

[5] Mihalcea, Rada & Corley, Courtney & Strapparava, Carlo. (2006). Corpus-based and Knowledge-based Measures of Text Semantic Similarity.. Proceedings of the National Conference on Artificial Intelligence. 1.

[6] U. L. D. N. Gunasinghe, W. A. M. De Silva, N. H. N. D. de Silva, A. S. Perera, W. A. D. Sashika and W. D. T. P. Premasiri, "Sentence similarity measuring by vector space model," *2014 14th International Conference on Advances in ICT for Emerging Regions (ICTer)*, Colombo, 2014, pp. 185-189, doi: 10.1109/ICTER.2014.7083899.

[7] B. (2017, August 8). *Compute sentence similarity using Wordnet - NLPFORHACKERS*. NLP-FOR-HACKERS. https://nlpforhackers.io/wordnet-sentence-similarity/

[8] Achananuparp, Palakorn & Hu, Xiaohua & Shen, Xiajiong. (2008). The Evaluation of Sentence Similarity Measures. 5182. 305-316. 10.1007/978-3-540-85836-2_29.

[9] Farouk, Mamdouh. (2019). Measuring Sentences Similarity: A Survey. Indian Journal of Science and Technology. 12. 10.17485/ijst/2019/v12i25/143977.

[10] Narayanan, S. (2020, September 3). *Semantic Similarity in Sentences and BERT - Analytics Vidhya*. Medium. https://medium.com/analytics-vidhya/semantic-similarity-in-sentences-and-bert-e8d34f5a4677

[11] Paul, R. (2020, December 28). *Location-Sensitive-Hashing-for-cosine-similarity - Analytics Vidhya*. Medium. https://medium.com/analytics-vidhya/location-sensitive-hashing-for-cosine-similarity-kaggle-nb-blog-f7b14523ced0

[12] Locality Sensitive Hashing (LSH) - Cosine Distance. http://ethen8181.github.io/machine-learning/recsys/content_based/lsh_text.html

[13] Zhang, Yin & Jin, Rong & Zhou, Zhi-Hua. (2010). Understanding bag-of-words model: A statistical framework. International Journal of Machine Learning and Cybernetics. 1. 43-52. 10.1007/s13042-010-0001-0.

[14] Brownlee, J. (2021). How to Develop a Deep Learning Bag-of-Words Model for Sentiment Analysis (Text Classification). Retrieved 25 January 2021, from https://machinelearningmastery.com/deep-learning-bag-of-words-model-sentiment-analysis/

[15] Maklin, C. (2019, August 05). Singular Value Decomposition Example In Python. Retrieved January 25, 2021, from https://towardsdatascience.com/singular-value-decomposition-example-in-python-dab2507d85a0