Report of Assignment - Big Data Mining Techniques

Requirement 1: Text Classification

Get to know the Data: WordCloud

At this part of the Assignment, before diving deeper to the text classification, we are called to explore the given data. Using the Python library 'WordCloud' and applying a minor pre-process to the train set, using English stopwords with some custom ones, the below results are extracted.

```
stop_words = STOPWORDS.update(['will', 's', 't', 'one', 'new', 'said', 'say', 'says', 'year'])
```

Classification Task

For the classification, initially I made a benchmark between SVM, Random Forests classifiers using TF-IDF as vectorizer. I didn't preprocess the data at this point (only transformed the text to lower case) because the only thing we want to extract is which classifier wins under what circumstances. So, I created 4 pipelines: TF-IDF + SVM, TF-IDF + Random Forest, TF-IDF + SVD + SVM & TF-IDF + SVD + Random Forest. The results were that TF-IDF + SVM has the best accuracy accross all the models but with the cost of high execution time (~4k secs.). Although, TF-IDF + SVD + Random Forest seems to be a promising model with ~94% F1-Score and low execution time.

Keeping that in mind, I further tuned this pipeline (TF-IDF + SVD + Random Forest), with some preprocessing and hyper-parameter tuning and finally the model achieved ~96% Accuracy. More specific the actions taken to boost this model were the following:

- Using the same stopwords from the WordCloud
- Repeat the titles 3 times for each article in order to add an extra weight to the words of the titles

- Convert the text to lowercase (obviously)
- Using a large amount of `max_features=50000` at the TF-IDF
- Use a dense "forest" of decision trees at the Random Forest Classifier (`n_estimators=1000`)

In order to conclude to this combination of hyperparameters I did approximately 30 different Cross Validation executions with different param combinations. The final hyperparam combination gave the best scores.

The table below shows the stats for each one of the models we referred above.

	SVM (BoW)	Random Forest (BoW)	SVM (SVD)	Random Forest (SVD)	My Method
Accuracy	0.97318	0.92976	0.94060	0.94711	0.96057
Precision	0.97196	0.93402	0.93743	0.94904	0.96049
Recall	0.96956	0.91313	0.92961	0.93312	0.95157
F1-Score	0.97074	0.92276	0.93339	0.94059	0.95586

Jupyter Notebooks:

WordCloud

Classification

BeatTheBenchmark

Requirement 2: Nearest Neighbor Search and Duplicate Detection

De-Duplication with Locality Sensitive Hashing

In order to detect duplicate data efficiently and accurate we should use the proper algorithms & tools. Some classic solutions of this problem is to calculate the similarity between the given texts using metrics such as Euclidean Similarity and Jaccard Similarity. At this task using these metrics I implemented 4 different algorithms, with the use of some libraries, that detect duplicates in different ways.

Before going any further to duplicate matching, first we should clean and preprocess the data. The preprocess stage consists of transforming to lower case the texts, remove any Arab, Chinese, non-ASCII characters, remove texts with less than 2 characters and remove punctuations such as 'dot', 'comma', etc.

Initially, we can detect duplicates naively, meaning that to calculate distances among all the elements of the datasets and choose the ones that have score higher than the defined (0.8 similarity). For estimating the Euclidean Similarity I used the `sklearn`'s function `cosine_similarity`. Calculating pairwise for all the elements of the test's set with the elements of train's set the algorithm found 1351 duplicates at 670 seconds.

Similarly, I implemented a custom function to calculate the Jaccard Similarity between 2 documents and use this method for all the elements of test's set to the elements of train's set. The function I implemented it is based on the formula of Jaccard Similarity, calculate the intersection of the 2 sets, calculate the union and finally return their quotient. Although, this algorithm is very slow, even using '`swifter`' package which parallelizes dataframe's apply method to all CPU cores. More specific it takes ~13K seconds and 430 duplicates were found.

In order to optimize further the above algorithms it is crucial to avoid the brute force logic. Using LSH family algorithms eventually this problem can be solved. LSH for a given document returns its approximate nearest neighbors and at this subset we can apply any other metric in order to find the true nearest neighbors. This is translated as duplicate neighbors to our problem.

So, for estimating LSH with Euclidean Similarity I have implemented a custom solution for LSH Random Projection which is based in the available literature and many stackoverflow/medium articles. It has been thoroughly tested in order to produce reliable results. This algorithm consists of 2 main parts. The train and the query. The train method initially generates k number of random vectors (or hyperplanes more accurately) generated via Gaussian Distribution. Using these hyperplanes the algorithm partitions the space and creates for each train document a bit vector that is assigned to one of the k hyperplanes. In that way we distribute the documents in separate sub areas. After that, when we have to query with a test document, we generate a bit vector for the test document and lookup the hypeplane that belongs to. If we match any of the hyperplanes, all the documents that belong to the matched hyperplane are returned to estimate the Euclidean Similarity with the test document. In that way we limit way too much the calculations that required to extract the duplicate documents.

Respectively, same logic is followed for the Jaccard Distance, but for this part I used the '`datasketch`' library. Datasketch provided Min-Hash LSH with Jaccard Similarity by default and there was need to proceed to custom solutions for this one. The key difference with LSH Random Projection and Min-Hash LSH is that for the later a big dictionary for all the words of documents is created. Using this dictionary the algorithm applies random permutations with the words in order to create fingerprints of each document. These fingerprints are approximations of the real documents and at querying are used to extract the similar ones using Jaccard Similarity. Also it is important to mention that the more permutations we do, the closer the approximations will be, at the cost of higher training execution times.

Finally, to sum all these up I provide a table that shows build/query times, number of duplicates for each one of the algorithms. It is very interesting how huge is the difference between Exact Jaccard and Min-Hash LSH Jaccard; LSH executes at ~300 seconds in contrast to 13K seconds of the Exact Jaccard.

STATS TABLE

Type	Build Time	Query Time	Total Time	# Duplicates	Parameters
Exact-Cosine	0	669.8874192237854	669.8874192237854	1351	-
Exact-Jaccard	0	13329.97571015358	13329.97571015358	430	-
LSH-Cosine	0.17445826530456543	1152.6631894111633	1152.837647676468	1213	k=1
LSH-Cosine	0.20483112335205078	611.8779292106628	612.0827603340149	1098	k=2
LSH-Cosine	0.1945638656616211	409.837290763855	410.0318546295166	942	k=3
LSH-Cosine	0.2286992073059082	206.18022441864014	206.40892362594604	842	k=4
LSH-Cosine	0.2292921543121338	99.041348695755	99.27064085006714	756	k=5
LSH-Cosine	0.2667248249053955	62.644378900527954	62.91110372543335	692	k=6
LSH-Cosine	0.29586219787597656	39.94454216957092	40.2404043674469	593	k=7
LSH-Cosine	0.276716947555542	31.290714740753174	31.567431688308716	515	k=8
LSH-Cosine	0.33685874938964844	26.838273525238037	27.175132274627686	464	k=9
LSH-Cosine	0.4166233539581299	24.317555904388428	24.734179258346558	406	k=10
LSH-Jaccard	343.6538245677948	3.3961427211761475	347.04996728897095	816	perm=16
LSH-Jaccard	447.6870336532593	4.637790679931641	452.3248243331909	619	perm=32
LSH-Jaccard	665.0936031341553	6.532337188720703	671.625940322876	673	perm=64

Some insights from the above table:

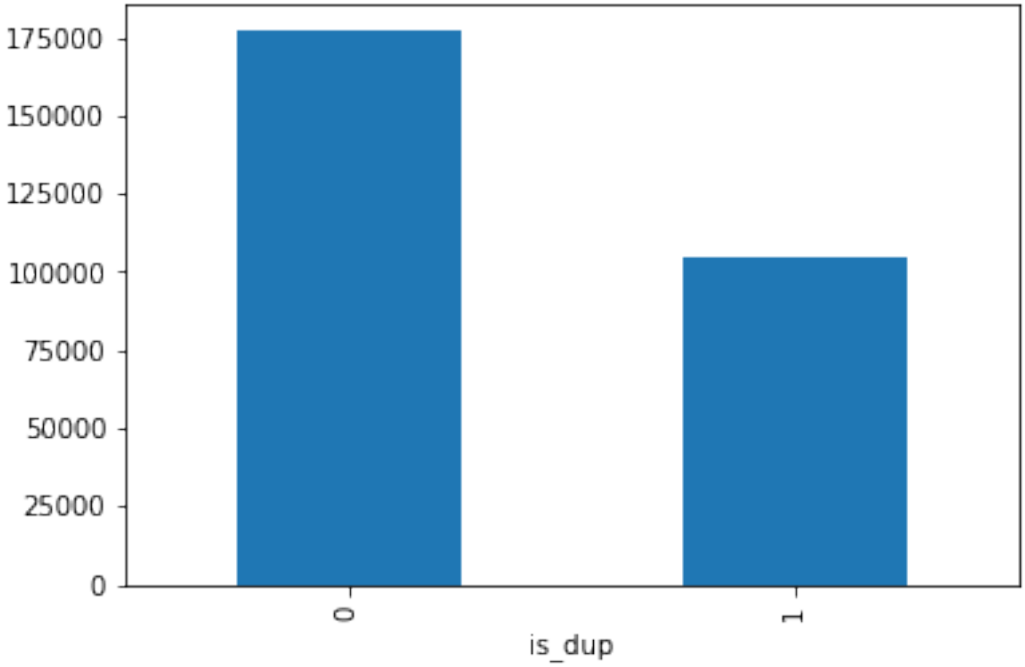
- As we increase the k at LSH-Cosine the duplicates tend to match the Exact Jaccard duplicates. Their Cosine Score should be very close to 1. This can be explained by that we limit the query documents to a space that contains a few train documents in order to calculate the distances between them. (In brute force we calculate 1 for N where N all the train dataset, in LSH we calculate 1 for M where M << N).
- Exact Jaccard is very CPU intensive because it extracts the Similarity direct from the text in contrast with the Exact Cosine which uses TF-IDF vectorizer. So, Exact Cosine uses vectorized calculations and that's why it is way faster than Jaccard.
- As we increase the permutations at LSH-Jaccard both build and query times are increased. That happens because we add extra overhead when building the index, adding more permutations for all the words. Likewise, for the query again we add extra overhead because the query document is transformed to MinHash object which uses the same number of permutations as the model in order to be compatible.

Jupyter Notebooks:

DeDuplication

Same Question Detection

At this part of the assignment it is required to detect duplicate questions/sentences, given a train dataset which consists of ~200K entries. These types of problems sometimes are hard to face because the data we have are pretty imbalanced (i.e. 80% percent of the data belong to one category and the rest 20% to the other). Some models can achieve high accuracy sometimes but the recall will be low, the elements of the second category will be wrongly classified due to lack of training data. Let's see if the training dataset is imbalanced.



Luckily the data is not very imbalanced. The 'no-duplicate' are ~70K more than the 'duplicate'. Although, we should keep that in mind to properly split the dataset at cross validation in order to extract reliable scores. This can be achieved by Stratified folds, the folds are made by preserving the percentage of samples of each class.

Now we can continue with the engineering of our model. The 'magic' recipe I used was to transform and extract a number of metrics that will seed an XGBoost Classifier. Before to conclude to this model initially I tested to fit/transform the texts to TF-IDF per question separately and stack them to one TF-IDF which was used as the train input of

the classifier. The problem I faced actually with this solution was that the model was over-fitting. Also it could detect easily the 'no-duplicate' items but it had low accuracy detecting the 'duplicate' items (low recall). Although with some tweaking at XGBoost I achieved almost 80% accuracy but that was meaning nothing due to low recall.

So I gave up this model and I started over with a new model, by generating features from the given data. The metrics generated are listed and explained further below.

The basics

- The length of question 1 & question 2
- The difference between the two question lengths
- The total characters of question 1 & question 2
- The difference between the two character lengths
- The number of words question 1 & question 2
- The number of common words

Some more intermediate

Using library fuzzywuzzy I managed to extract some more metrics about the similarity ratio between questions. This library uses Levenshtein Distance in order to calculate the differences among the questions (similar work as the de-duplication part of this assignment)

- Fuzz Simple Ratio, the ratio of similarity between the questions
- Fuzz Partial Ratio, the ratio of the most similar substring
- Fuzz Token Sort Ratio, the measure of the sequences' similarity between 0 and 100 but sorting the token before comparing.
- Fuzz Token Set Ratio, same as above but treating tokens as a set
- Fuzz Partial Token Sort Ratio, the ratio of the most similar substring as a number between 0 and 100 but sorting the token before comparing.
- Fuzz Partial Token Set Ratio, same as above but treating tokens as a set

Now we have engineered a good starting number of features. A preliminary 5-fold cross validation run (not included in the submitted notebook) gave the following results:

- Accuracy: 0.734332368
- Precision (macro): 0.715948086
- Recall (macro): 0.718021134
- F1-Score (macro): 0.71691373

The scores are a bit low but we can increase them by generating even more features. Let's dive deeper to feature generation by extracting distances between the sentences.

- Cosine Distance
- Cityblock Distance
- Jaccard Distance
- Canberra Distance
- Euclidean Distance
- Minkowski Distance
- Braycurtis Distance
- Skew & Kurtosis stats
- (Word2Vec) The Word Mover's Distance between two questions. WMD use word embeddings to calculate the distance so that it calculate even though there is no common word. The assumption is that similar words should have similar vectors.

For all the above I have used the `scipy`'s implemented methods and Word2Vec `gensim` library.

Word2Vec

Some important details about Word2Vec. Using this framework with the pre-trained corpus 'GoogleNews-vectors-negative300.bin.gz' we generate a vector space with each word in the corpus being assinged a corresponding vector in space. With the use of this we can extract the context of each question and compare them in order to extract their similarity. So, by computing the vectors for all question1 and question2 we apply to them all the distance functions that referred above.

The link of the pre-trained corpus is attached to the notebook

Summarizing all these up, the final training data consists of a dataframe that has only numerical data. The only that is left is a proper classifier that manipulates and classifies all these data properly.

XGBoost

XGBoost is very well known machine learning algorithm and the last years has been applied to many applications, including duplicate detection problems. Many data scientists prefer to use it because its speed and the accuracy that provides. Briefly, this algorithm is an implementation of gradient boosting decision trees. Therefore, I choose this algorithm in order to learn how to use it and also to see how it performs. Also I have made a comparison between XGBoost and Random Forests in order to see the performance differences (not included in the report you can find it at my notebook at github page, link is provided below) (experiment after the deadline).

Stats

Eventually, XGBoost with the new features has increased performance with relatively good accuracy and recall. The table below shows the stats after a Stratified 5-fold Cross Validation.

The parameters of XGBoost for this execution were:

```
max_depth=80, n_estimators=500, learning_rate=0.1, objective='binary:logistic',
eta=0.3
```

	XGBoost
Accuracy	0.76755
Precision	0.75195
Recall	0.75799
F1-Score	0.75446

Maybe with different parameters could be achieved higher scores, but due to lack of time I wasn't able to tweak the hyperparameters even more. Although, I will test this after the deadline and in case of finding a better hyperparameter combo I will update the notebook on my repository.

Jupyter Notebooks:

DuplicateDetection