ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

12 Φεβρουαρίου 2023

Γιώργος Γκρέκας & Πολλάλη Μαρία-Αγγελική

Τεχνικές Ανάλυσης Δεδομένων Υψηλής Κλίμακας

Διδάσκων : Γουνόπουλος Δ.

1 Requirement 1: Text Classification

1.1 Preprocessing

In order to implement the machine learning and AI techniques for the classification of the articles in the four categories: business, entertainment, healthy and technology we did some reprocessing steps. Specifically, we implement the following:

1. **Decode ASCII:** ASCII (American Standard Code for Information Interchange) is a character encoding standard that assigns unique numbers to each letter, digit, and other symbol used in written text.
2. **Lower casing:** Converting a word to lower case (eg. NLP to nlp). We did that because words like Book and book mean the same but when not converted to the lower case those two are represented as two different words in the vector space model (resulting in more dimensions).
3. **HTML tags:** Removing all HTML tags (
) with empty spaces.
4. **Stop words** stop words occur in abundance, hence providing little to no unique information that can be used for classification or clustering.

Also, we created a new column which is a combination of Title and Content. We did a lot of tries with different weights in both columns (Title and Content) and we end up with the following formula:

$$\textbf{df['Combined']} = \textbf{3*(df['Title'] + ' ') + df['Content']}$$

Finally, we used **TF-IDF**. TF-IDF which means Term Frequency and Inverse Document Frequency, is a scoring measure widely used in information retrieval (IR) or summarization. TF-IDF is intended to reflect how relevant a term is in a given document.

Note: There are more steps in the data reprocessing for NLP like: tokenization, stemming, lemmatization and so on .We decided not to include them because the pre-processing took a lot of time with them and he achieved a good accuracy without adding them to the pre-processing pipeline.

1.2 Models, features and parameters selected

In order to classify the test dataset(unseen) we train the below 5 models. All models, since it's a classification problem are evaluated in accuracy, precision, recall and f1-score.

1. Linear SVC: Linear Support Vector Machine
2. KNN: K-Nearest Neighbors
3. Random Forest
4. XGBOOST
5. *Neural Networks*

Linear SVC:
We fine tuned the SVC regarding 2 parameters:

- Tol: Tolerance for stopping criteria.
- Random State: Controls the pseudo random number generation for shuffling the data for the dual coordinate descent

After a lot of tries we ended up with tol=1e-5 and random state=42. In this way the accuracy that we had in the train dataset was the following:

accuracy	precision	recall	F1-score
0.976	0.9745	0.973	0.9737

Note: For all the models we tried SVD. SVD is used in latent semantic analysis(LSA). Latent Semantic Analysis is a technique for creating a vector representation of a document. Vector representation allows to do handy things like classifying documents to determine which of a set of known topics they most likely belong to. In LSA the first step is TF-IDF. SVD is the second step. SVD takes TF-IDF one step further. SVD is used to perform dimensionality reduction on the TF-IDF vectors generated in step 1. We will presented the results of SVC with SVD for this model but not for the others since the results were better without using it.

accuracy	precision	recall	F1-score
0.8894	0.8826	0.8717	0.8767

KNN:
We fine tuned the KNN regarding neighbors parameter. Specifically, we tried 3, 5, 10, 15 and 20. We ended up using 10 neighbors. In this way the accuracy that we had in the train dataset was the following:

accuracy	precision	recall	F1-score
0.9725	0.9703	0.9691	0.9697

Random Forest:
We fine tuned the Random Forest regarding 3 parameters:

- n Estimators: The number of trees in the forest.
- Max Depth: The maximum depth of the tree.
- Random State: Controls both the randomness of the bootstrapping of the samples used when building trees nd the sampling of the features to consider when looking for the best split at each node.

After a lot of tries we ended up with n estimatos=100, max depth=18 and random state=42. In this way the accuracy that we had in the train dataset was the following:

accuracy	precision	recall	F1-score
0.7569	0.866	0.6749	0.7228

XGBoost:
We fine tuned the XGBoost regarding 2 parameters:

- Objective: Specify the learning task.
- Random State: For the suffling.
- Max Depth: Maximum depth of a tree.

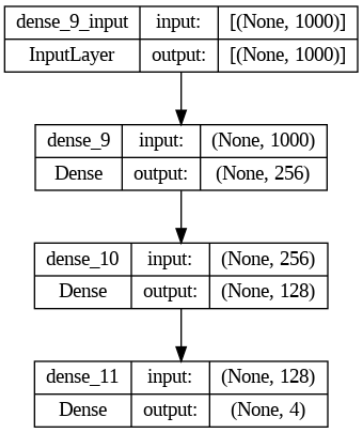
After a lot of tries we ended up with objective = multi:softprob, random state = 42 and max depth = 18. In this way the accuracy that we had in the train data set was the following:

accuracy	precision	recall	F1-score
0.7578	0.874	0.6755	0.7240

Note: We did the hyper parameter tuning a little bit manually because the using of grid search took a lot of time.

Neural Network:
Neural networks are the most computationally expensive models as their demands grow exponentially by adding more layers and more units within a layer. So at first we restricted our number of features to 1000. This was done by keeping only the tokens with a large (top 1000) frequency. Our model is a sequential neural network with 4 layers, the input layer, two hidden layers consisting of 256 and 128 nodes respectively and the output layer of 4 nodes, equal to the number of classes. Hidden layers are activated with the ReLU function and the output layer with the Softmax function as we need to perform a multiclass classification. Also we chose the sparse categorical entropy because our data are vectorized using tfidf. The number of epochs was chosen to be 3 as after experimenting we noticed that the accuracy metric does not increase.

accuracy	precision	recall	F1-score
0.9923	0.992	0.9909	0.9914



2 Requirement 2: Nearest Neighbor Search with Locality Sensitive Hashing

Brute Force Cosine (one vs all):
For this task we chose to convert our categorical data to numerical using tf-idf vectorization. Brute Force algorithm performed full classification in a relatively good amount of time (35.35 s.)

LSH Cosine (random projection):
Using the same tf-idf vectorization, we implemented the bi-partition of our space with 2,3,5,10 and 100 hyper-planes. For n hyper-planes there are 2^n possible buckets for an every element. So the expected behaviour of our experiments is that for a higher number of hyper-planes there will be higher built time. On the contrast, the more the buckets, the less elements would end up in it, so query time would decrease significantly but accuracy would be dropping too. We have to also note that we chose our fingerprints to be a string sequence of bits in order to avoid overflow as 2^100 is a very large number to be an efficient bucket label. In the above matrix we can see the results.

Brute Force Jaccard (one vs all):
For this task we chose to convert our categorical data to numerical using CountVectorizer, basically what we did was to create a vector with dimensions equal to number of unique words and one's in the positions where every word was present in each document. The reason behind this implementation is the Jaccard metric which does not quantify the frequency, but the existence of similar tokens. What we notice is that a simple query for the brute force algorithms takes around 10 seconds. So in order to compute all test set queries we would need 27 hours.

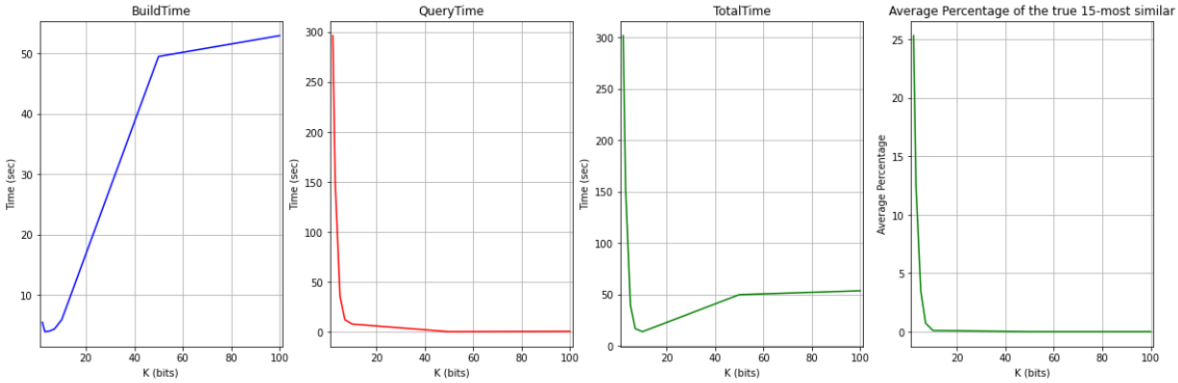
LSH Jaccard (MinHash):
Using the same vectorization as Brute Force Jaccard we implemented the bi-partition of our space with MinHash. Minhash is a techique based in the expected (and not accurate) Jaccard similarity. Given a vector from the test set, using this procedure it can find in a small amount of time all vectors in the train set with expected Jaccard similarity over a user-defined threshold. We chose our threshold to be 0.4 in order to get rid of empty buckets. The higher the threshold the smaller the number of train vectors in a bucket. Also the expected behaviour is that for a higher number of permutations the expected jaccard similarity will tend to the true jaccard similarity and its behaviour will be equal (depending on the threshold) to the Brute Force's one.

Note 1:
Random projection fills the buckets before the query, so for the first task, the knn initialization process could be a part of the built time, shrinking query time. On the contrast for the second task, each bucket is created when a query vector comes, so the knn initialization is a part of the query time.

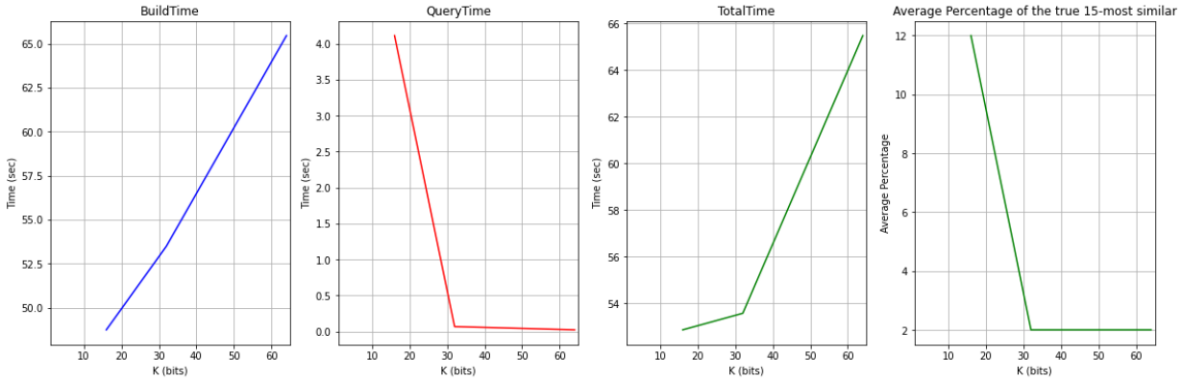
Note 2:
For the second task we used a sample of one third in order to estimate the results.

Type	BuiltTime	QueryTime	TotalTime	Perc of true K	Parameters
Brute-Force-Jaccard	0	27h	27h	100	-
LSH-Jaccard	13h	1h	14h	12	P:16,thr:0.4
LSH-Jaccard	14.5h	1m	14.5h	2	P:32,thr:0.4
LSH-Jaccard	18h	20s	18h	2	P:64,thr:0.4
LSH-Jaccard	14.7h	9s	14.7h	0.7	P:16,thr:0.6
LSH-Jaccard	15.5h	10s	15.5h	0.7	P:32,thr:0.6
LSH-Jaccard	18.6h	10s	18.6h	0.7	P:64,thr:0.6
Brute-Force-Cosine	0	35s	35s	100	-
LSH-Cosine	5.5s	296s	301s	25.3	K:2
LSH-Cosine	4s	149s	153s	12.5	K:3
LSH-Cosine	3.9s	35s	40s	3.5	K:5
LSH-Cosine	4.35s	12s	17s	0.75	K:7
LSH-Cosine	5.9s	8s	14s	0.1	K:10
LSH-Cosine	49.5s	0.5s	50s	0	K:50
LSH-Cosine	53s	0.8s	53.8s	0	K:100

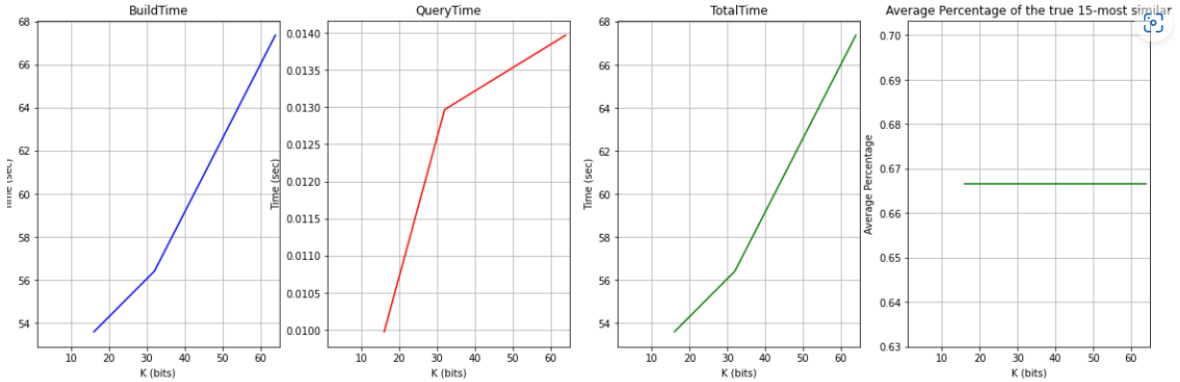
Cosine



Jaccard with thr:0.4



Jaccard with thr:0.6



Note 3:
We notice that for more complicated procedures (higher number of permutations, higher hash functions) we have a significant increase in our build time with no improvement on accuracy or query time.

3 Requirement 3: Time Series Similarity

For this exercise we implement the algorithm Dynamic Time Warping(DTW) in order to compute the similarities between time series of different time resolutions. We implement the function from scratch. We consider as s and t the two arrays that are correspond to the rows of series a and series b. The logic is described in the following steps:

1. Initialize variables: The first two lines of the function extract the number of rows (i.e. the length) of the input arrays s and t, and assign the values to variables n and m. Then, a two-dimensional array DTW of size (n+1, m+1) is created, filled with the value np.inf (i.e. positive infinity).

2. Set the starting point: The value at the first cell of the DTW array, DTW[0, 0], is set to 0. This represents the starting point of the DTW calculation.

3. Loop through arrays: The next two nested for-loops are used to iterate through the arrays s and t. The outer loop i goes from 1 to n+1 (inclusive), and the inner loop j goes from 1 to m+1 (inclusive).
4. Calculate cost: At each iteration of the inner loop, the cost variable is calculated as the Euclidean distance between the current elements of s and t, i.e. np.linalg.norm(s[i-1]-t[j-1]). This represents the cost of matching the two elements.
5. Update DTW array: The value of DTW[i, j] is then updated as the minimum of the values at the three cells above it: DTW[i-1, j], DTW[i, j-1], and DTW[i-1, j-1], plus the cost calculated in step 4. This implements the core dynamic programming step of DTW.
6. Return result: Finally, the function returns the value at the last cell of the DTW array, DTW[n, m], which represents the DTW distance between the two input arrays.

So, in order to implement this in python we build the following function:

```
def DTWDistance(s, t):
    n, m = s.shape[0], t.shape[0]
    DTW = np.full((n+1, m+1), np.inf)
    DTW[0, 0] = 0

    for i in range(1, n+1):
        for j in range(1, m+1):
            cost = np.linalg.norm(s[i-1]-t[j-1])
            DTW[i, j] = cost + min(DTW[i-1, j],
                                   DTW[i, j-1],
                                   DTW[i-1, j-1])
    return DTW[n, m]
```

The whole procedure took 74.16635239631665 minutes.

The output file is a csv with two columns: Id and distance. You could find it in the Kaggle page.