Data Science and Information Technologies

Konstantina Alliou 7115152400011

Nikolaos Charisis 7115152400008

INTRODUCTION

This project aims to provide a structured approach to applying data mining techniques, encompassing data collection, preprocessing, conversion, application of algorithms, and evaluation. The implementation is conducted using Python, leveraging the SciKit-Learn library to facilitate various machine learning tasks.

The dataset consists of articles classified into four categories: **Business, Entertainment, Health, and Technology**. The project is divided into three main sections, each addressing a specific data mining challenge:

1. **Text Classification** – This section focuses on classifying articles into their respective categories using machine learning techniques.
2. **Nearest Neighbor Search with Locality Sensitive Hashing (LSH)** – This section explores nearest neighbor search, comparing traditional methods with LSH to optimize search efficiency.
3. **Time Series Similarity (Dynamic Time Warping)** – This section implements Dynamic Time Warping (DTW) to measure similarity between time series data, which is particularly useful in applications involving sequential patterns.

Through these tasks, the project provides hands-on experience in implementing and evaluating data mining techniques, reinforcing key concepts in machine learning and data analysis. The dataset was available at the address:https://www.kaggle.com/competitions/bigdata2024classification/data

Part 1: Text classification

QUESTION 1:

In this classification task, we implement two machine learning models for the entirety of the two datasets Support Vector Machines (SVM) and Random Forestto categorize articles into one of four categories: Business, Entertainment, Health, or Technology.

The feature extraction method used is Bag of Words (BoW), which represents text data as numerical vectors based on word frequency. For a small fraction of the datasets we have added KNNclassifier with Jaccard. To ensure robust evaluation, we apply 5-fold Cross Validation, measuring model performance using accuracy as the evaluation metric. This approach helps assess the effectiveness of each model-feature combination, providing insights into their classification capabilities.

Evaluation Results for 1% :

Statistic Measure	SVM (BoW)	Random Forest (BoW)	KNN With Jaccard
Accuracy	90.06 %	85.14 %	83.08 %

According to our results the best classification method is SVM (BoW).For 1% of our dataset the Runtime for SVM is 0.15 sec, the Runtime for the Random Forest is 6.54 secs and lastly the Runtime for the KNN is 275.00 secs. Due to the Runtimes we can deduce that the KNN classifier is significant slower than the other methods even for the 1% of our data. So this method is disqualified for the final prediction as it would take an inappropriate amount of time to predict for the entire dataset. However, we tried the KNN also for the 10 % of the dataset.

Evaluation Results for 10% :

Statistic Measure	SVM (BoW)	Random Forest (BoW)	KNN With Jaccard
Accuracy	95.50%	89.20 %	71.10 %

According to our results the best classification method is again SVM (BoW).Additionally, the KNN took us approximately 7 hours. So once again this method proved to be too slow to have any practical uses.

Having clarified that KNN is unsuitable for large volumes of data we proceed with just SVM (BoW) and Random Forest (BoW) for the complete dataset .

Evaluation Results for 100% :

Statistic Measure	SVM (BoW)	Random Forest (BoW)
Accuracy	97.42 %	93.42 %

In the given table we can observe the results for the complete dataset using 5-Fold Cross- Validation method for SVM (BoW) and Random Forest (BoW) .Apparently, the runtime for SVM (BoW) is 28.15 secs and the runtime for the Random Forest is 4020.27 secs. It is easy to see that ,due to the above results , SVM (BoW) is the best method not only in terms of accuracy but also in terms of Runtime.

The required output prediction can be found in the Results.zip file .

Part 2: Nearest Neighbor Search with Locality Sensitive Hashing

Question 2:

We are going to explore the effect of the shingles' size to our results in different metrics (Build Time , Query Time , Total Time and Fraction).

In the folllowing tables the threshold is 0.5 and we are experimenting on 2% of the datasets.

Shingling forsize k=2 :

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**79.94 sec**	**79.94 sec**	**100%**	---------
LSH-Jaccard	**6.85 sec**	**3.19 sec**	**10.04 sec**	**55%**	**Perm=16**
LSH-Jaccard	**7.17 sec**	**3.45 sec**	**10.62 sec**	**72%**	**Perm=32**
LSH-Jaccard	**7.81 sec**	**3.87 sec**	**11.68 sec**	**86%**	**Perm=64**

Shingling forsize k=3:

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**222.72 sec**	**222.72 sec**	**100 %**	**---------**
LSH-Jaccard	**19.33 sec**	**8.17 sec**	**27.5 sec**	**25 %**	**Perm=16**
LSH-Jaccard	**19.18 sec**	**8.22 sec**	**27.4 sec**	**10%**	**Perm=32**
LSH-Jaccard	**19.94 sec**	**8.60 sec**	**28.54 sec**	**13%**	**Perm=64**

Shingling forsize k=7:

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**357.91 sec**	**357.91 sec**	**100 %**	**---------**
LSH-Jaccard	**33.71 sec**	**14.59 sec**	**48,3 sec**	**1 %**	**Perm=16**
LSH-Jaccard	**34.23 sec**	**14.73 sec**	**48.96 sec**	**~ 0 %**	**Perm=32**
LSH-Jaccard	**36.10 sec**	**15.55 sec**	**51,65 sec**	**~ 0%**	**Perm=64**

Judging by our results we notice the following:

- The size of shingles affects the build time and query time of all methods. The higher the k the longer the times (linearly associated).
- The size of shingles affects fraction of all non-Brute – Force methods. The higher the k the lower the fraction.

Based on the aforementioned the best k is the k =3.

In the following tables we explore the threshold in regards to the various times and the fraction given the k=3 and we are working into our 2% of our data.

For the threshold =0.6

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**229.65 sec**	**229.65 sec**	**100 %**	**---------**
LSH-Jaccard	**18.84sec**	**8.09sec**	**26.93sec**	**8 %**	**Perm=16**
LSH-Jaccard	**19.01sec**	**8.19sec**	**27.2sec**	**3 %**	**Perm=32**
LSH-Jaccard	**21.07sec**	**8.58sec**	**29.65sec**	**1%**	**Perm=64**

For threshold=0.5

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**222.72sec**	**222.72 sec**	**100 %**	**---------**
LSH-Jaccard	**19.33 sec**	**8.17 sec**	**27.5 sec**	**25 %**	**Perm=16**
LSH-Jaccard	**19.18 sec**	**8.22 sec**	**27.40 sec**	**10 %**	**Perm=32**
LSH-Jaccard	**19.94 sec**	**8.60 sec**	**28.54 sec**	**13 %**	**Perm=64**

For threshold =0.4

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**242.29 sec**	**242.29 sec**	**100 %**	**---------**
LSH-Jaccard	**19.21 sec**	**8.46 sec**	**27.67 sec**	**40 %**	**Perm=16**
LSH-Jaccard	**19.99 sec**	**8.47 sec**	**28.46 sec**	**31 %**	**Perm=32**
LSH-Jaccard	**20.41 sec**	**9.01 sec**	**29.42 sec**	**14%**	**Perm=64**

To sum up, we notice that the higher the similarity threshold the lower the fraction of the true K-most similar documents that the LSH method also returns. We support that these observations are expected as we had to deal with real world articles. This means that the majority of our data can't be 'too similar' because that would be plagiarism in real life which can be considered a crime. So taking that into consideration we decided the appropriate threshold is 0,4.

We were unable to evaluate themethods using the complete datasets due to time restrictions. However, we have gathered results using numerous subsets of various sizes. Therefore, it is safe to assume that the results that we obtained approximate the entire datasets sufficiently.

Based on our previous results we set k=3 and threshold=0.4.

The results are shown in the table below:

Subset size 1% of the original dataset :

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**43.99 sec**	**43.99 sec**	**100 %**	**---------**
LSH-Jaccard	**8.29 sec**	**3.44 sec**	**11.73 sec**	**29%**	**Perm=16**
LSH-Jaccard	**8.20 sec**	**3.50 sec**	**11.70 sec**	**29%**	**Perm=32**
LSH-Jaccard	**8.56 sec**	**3.63 sec**	**12.19 sec**	**11%**	**Perm=64**

Subset size 2% of the original dataset :

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**242.29sec**	**242.29 sec**	**100 %**	**---------**
LSH-Jaccard	**19.21 sec**	**8.46 sec**	**27.67 sec**	**40 %**	**Perm=16**
LSH-Jaccard	**19.99 sec**	**8.47 sec**	**28.46 sec**	**31 %**	**Perm=32**
LSH-Jaccard	**20.41 sec**	**9.01 sec**	**29.42 sec**	**14%**	**Perm=64**

Subset size 5% of the original dataset :

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**1089.39 sec**	**1089.39 sec**	**100 %**	**---------**
LSH-Jaccard	**39.60 sec**	**17.72 sec**	**57.32 sec**	**32 %**	**Perm=16**
LSH-Jaccard	**41.39 sec**	**18.05sec**	**59.44 sec**	**31 %**	**Perm=32**
LSH-Jaccard	**42.88 sec**	**18.36 sec**	**61.24 sec**	**13%**	**Perm=64**

Subset size 7.5 % of the original dataset :

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**2485.85 sec**	**2485.85 sec**	**100 %**	**---------**
LSH-Jaccard	**59.57sec**	**27.65sec**	**87.22 sec**	**33 %**	**Perm=16**
LSH-Jaccard	**63.29sec**	**29.52 sec**	**92.81 sec**	**32 %**	**Perm=32**
LSH-Jaccard	**65.06 sec**	**28.01 sec**	**93.07 sec**	**14%**	**Perm=64**

Subset size 10% of the original dataset:

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**4549.55 sec**	**4549.55 sec**	**100 %**	**---------**
LSH-Jaccard	**80.88 sec**	**38.44sec**	**119.32 sec**	**34 %**	**Perm=16**
LSH-Jaccard	**81.55sec**	**37.78sec**	**119.33 sec**	**32 %**	**Perm=32**
LSH-Jaccard	**87.43sec**	**38.03sec**	**125.46 sec**	**14%**	**Perm=64**

Subset size 12,5% of the original dataset :

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**9237.24 sec**	**9237.24 sec**	**100 %**	**---------**
LSH-Jaccard	**122.70sec**	**61.34sec**	**184.04 sec**	**44 %**	**Perm=16**
LSH-Jaccard	**123.79sec**	**58.61 sec**	**182.40 sec**	**34 %**	**Perm=32**
LSH-Jaccard	**132.93sec**	**55.08 sec**	**188.01 sec**	**17%**	**Perm=64**

Subset size 20% of the original dataset :

Type	BuildTime	QueryTime	Total Time	Fraction	Parameters
Brute-Force Jaccard	**0**	**19180.90 sec**	**19180.90 sec**	**100 %**	**---------**
LSH-Jaccard	**163.86sec**	**84.22sec**	**248.08 sec**	**38 %**	**Perm=16**
LSH-Jaccard	**167.70sec**	**85.03 sec**	**252.73 sec**	**35 %**	**Perm=32**
LSH-Jaccard	**178.82sec**	**78.91 sec**	**257.73 sec**	**17%**	**Perm=64**

With the above results from the tables we can see that the LSH method heavily relies on the permutations parameter.

Respectively,

For perm=16:

Average fraction=35.71

For perm=32:

Average fraction=32

For perm=64:

Average fraction=14.28 ,

It is noticeable that the higher the permutation goes the lower the fraction is. Regarding the time aspect , we can notice that the more permutations we have more time is required to build the LSH model. However , the query time is diminished as the permutation increase.

Due to the nature of the problem and the data the above tendency holds true for the entire dataset.

All in all , and taking everything into consideration we can safely assume that the LSH methods are faster than Brute Force KNN. On the other hand, the Brute Force method is considerably more accurate as it is indicated from the fraction values, something that is to be expected .

Part 3: Time Series Similarity

Question 3: Dynamic Time Warping Implementation

Changing gears, we move on with the implementation of the algorithm Dynamic Time Warping(DTW) in order to compute the similarities between time series of different time resolutions. We implemented the DTW algorithm from scratch and utilizing Dynamic Programming we were able to calculate the required distances outputting them in the dtw.csv file. For each entry of columns series_a and series_b we calculated the distance between them using DTW Euclidean distance .

The whole process of the Dynamic Time Warping(DTW) took 741.3972 seconds.

Code Documentation/ Design Choices/Resources

First and foremost, it is worth mentioning that each question has its own dedicated Jupyter Notebook which implements the required algorithms and methods. So, part1.ipnyb answers part1, part2.ipynb answers part2 and part3.ipynb answers part3. In the Results.zip file, you can find the required .csv output files.

Regarding the implementation of question 1, a handful of design choices have been made:

- The testing of the kNN method happens only below a certain fraction of the subsets of the train and test sets. The reason behind that is that our experiments have shown that even for miniscule subsets (e.g. 1% of the originals), kNN is tremendously slower without being actually better than all of the other methods in question. So, when it comes to comparing based on larger fractions and the entirety of the datasets, we elect to ignore it.
- For the vectorization of our data, the TfidVectorizer of sklearn has been used. That’s because the weighting of the words that Tfid utilizes is really impactful to our text classification problem.
- For the SVM classifier, we use sklearn’sLinearSVC classifier, since per sklearn’s documentation it is the optimal SVM for out text classification task. This becomes immediately apparent once combine it with our choice for the vectorizer, TfidVectorizer. This combo can reach very high accuracies in very small times, as we have already demonstrated in the tables of the previous sections

- To be able to use sklearn's kNN classifier, we had to implement the Jaccard Distance metric. A point of interest here is the transition between the sparse matrices that the vectorizer returns and the numpy arrays used to calculate the non-zeros. Since the data structure of a sparse matrix doesn't store data in every entry of the matrix, the usual methods of iterations and traversal don't apply. To overcome that, we had to first convert the sparse matrix to an numpy array and then proceed with applying the proper numpy methods
- In general, to reach our conclusions, we have conducted various experiments using subsets of various sizes. For the SVM and RandomForest methods, this is no longer need. However, for the comparison with the kNN algorithm, this is the only way of yielding any type of results in a reasonable time. So, the code remains unchanged but for the last two lines in respective part, where we create the subsets.

In regards to question 2's implementation, it is worth mentioning the following:

- We have decided to approach the problem by utilizing document shingling. The shingling process, instead of taking into account k-grams of words, takes into consideration k-grams of characters. So in that aspect, we diverge from the recommended approach
- Before we shingle the documents, we first preprocess them by removing punctuation, stop words and case sensitivity.
- The aforementioned occur as an attempt to emulate the example given by datasketch's documentation on LSH.
- The similarity now becomes just the intersection divided by the union, with the two operations being already available by python's Set data structure
- In reality, however, we chose to work with the Jaccard Distance. If we ever wanted to utilize the Similarity as is, we just need to reverse the sorting of the respective neighbors list.

Respectively, for question 3, some points of interest are:

- As a cost, we use the Euclidean distance for scalars, that being the absolute value of their difference. Obviously, if we alter the cost function we get different results. The algorithm, however, would still work as intended
- A special case is the first cell of the dtw_matrix. Arbitrarily, we have set its value to the result of the cost function for the first point of the pair of time series we are examining at a given point
- A couple more special cases occur for the first row and for the first column respectively. We take special care of those in the lines outlined by the appropriate comments

Conclusion

This project was all about diving into data mining and testing out different techniques to see what works best. For text classification, SVM (BoW) came out on top, `beating` other methods in both speed and accuracy. When it came to searching for similar data, LSH proved to be way faster than Brute-force methods while still keeping decent accuracy. And for time series analysis, DTW did a solid job of spotting similarities between patterns. In the end, this project gave us a hands-on look at machine learning in action, highlighting both the strengths and limitations of different algorithms in real-world scenarios.

RESOURCES FOR THE PROJECT IMPLEMENTATION:

- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
- https://scikit-learn.org/1.5/auto_examples/text/plot_document_classification_20newsgroups.html#sphx-glr-auto-examples-text-plot-document-classification-20newsgroups-py
- https://ekzhu.com/datasketch/lsh.html
- https://numpy.org/doc/stable/index.html
- https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html
- https://www.nltk.org/
- https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
- https://docs.scipy.org/doc/scipy/reference/sparse.html

PC SPECIFICATIONS:

- CPU: AMD Ryzen 5 4600G with Radeon Graphics
- GPU: AMD Radeon HD 7700 series
- RAM: 16GB DDR5

Thank you very much!

Konstantina Alliou & Nikolaos Charisis