NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

An Interdepartmental program between the NKUA Departments of Physics and of Informatics and Telecommunications with specialization in Control and Computing

Big Data Mining Techniques

DIMITRIOS TSINTSILONIS

AM: 7110132300202

LOUKAS DROSOS

AM: 7110132400201

Supervisor Professor:

GOUNOPOULOS DIMITRIOS

Athens 2024 – 2025

Contents

Introduction

The main objective of this project documentation is to understand the basic steps of the process followed for applying data mining techniques, namely: collection, preprocessing / cleaning, conversion, application of data mining techniques and evaluation using the Python programming language. The project consists of three tasks related to categorization, nearest neighbors and trajectory similarity.

This documentation consists of three chapters. Chapter 1 is related to the text classification of news articles using the classification methods of Support Vector Machines and Random Forests. The main objective of Chapter 2 is to speed up the K-NN classification method of a given train set file with small text, using the brute-force method and Locality Sebsitive Hashing. Finally, in Chapter 3 we implement the algorithm Dynamic Time Warping in order to compute the similarities between time series of different time resolutions.

1. Text Classification

This task involves the classification of news articles based on textual content. There are 4 categories of articles: Business, Entertainment, Health and Technology. The dataset is stored in CSV files, where fields are separated by the ',' character. There are two datasets:

1. **train_set.csv** (111,795 entries): This file is used to train the classification model and consists of the following fields:
   - **Id**: A unique identifier for each article.
   - **Title**: The article's headline.
   - **Content**: The full text of the article.
   - **Label**: The category assigned to the article.
2. **test_set.csv** (47,912 entries): This file contains new, unseen data for prediction. It includes the same fields as the training set, except for the **Label** field, which needs to be predicted using classification algorithms.

For this task we will use and compare two classification methods, Support Vector Machine and Random Forests.

Support Vector Machine (SVM) is a supervised learning algorithm commonly used for classification and regression tasks. While it can handle regression, it is particularly effective for classification problems.

The key idea behind the SVM algorithm is to find the hyperplane that best separates two classes by maximizing the margin between them. This margin is the distance from the hyperplane to the nearest data points (support vectors) on each side.

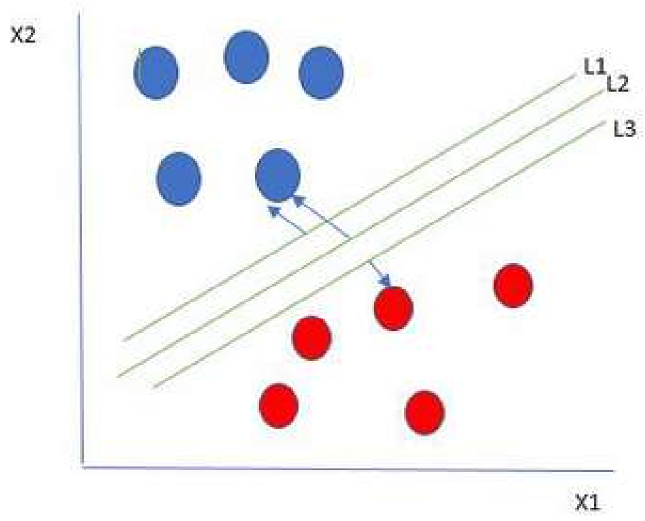


Figure 1.1. *Multiple hyperplanes separate the data from two classes*

The best hyperplane is the one that maximizes the distance between the hyperplane and the nearest data points from both classes. This ensures a clear separation between the classes. So, from Figure 1.1 the best hyperplane is L2.

Random Forest algorithm is a powerful tree learning technique that is widely used for classification and regression task. It is a type of classifier that uses many decision trees to make predictions by taking different random parts of the dataset to train each tree and then it combines the results by averaging them. This approach helps improve the accuracy of predictions.

The evaluation of these classification methods will be done using the Bag of Words model. The bag-of-words model (BoW) is a model of text which uses an unordered collection (a "bag") of words. BOW is commonly used in methods of document classification where, for example, the (frequency of) occurrence of each word is used as a feature for training a classifier. It disregards word order (and thus most of syntax or grammar) but captures multiplicity.

The performance of every model and feature combination will be evaluated with 5-fold Cross Validation using Accuracy. Accuracy is a common metric used to evaluate classification models and is the number of correct predictions divided by the total number of predictions.

Cross-validation is a technique used to assess the performance of a model by dividing the dataset into multiple parts. In 5-fold Cross-Validation, the dataset is split into five equal parts (folds).

The process follows these steps:

1. Divide the dataset into five equal subsets (folds).
2. Train the model on four of the subsets and test it on the remaining one.
3. Repeat this process five times, with a different subset being used as the test set each time.
4. Compute the average accuracy across all five iterations to get a more reliable performance estimate.

By using 5-fold Cross-Validation with Accuracy, we ensure that the model's performance is robust, unbiased, and well-tested across different subsets of the data. It helps identify the best combination of models and features for achieving high classification accuracy.

In our code, first we import the required libraries mentioned below:

- pandas → Used for handling datasets (reading CSV, data manipulation).
- re → Used for regular expressions (text processing).
- numpy → Supports numerical computations and array operations.
- nltk → Natural Language Toolkit, used for text processing.

- matplotlib.pyplot → Used for visualizing data.
- sys → Provides system-related functionalities.
- word_tokenize → Splits text into words.
- stopwords → Provides a list of common words to remove (e.g., "the", "is", "and").
- WordNetLemmatizer → Reduces words to their base form (e.g., "running" → "run").
- Pipeline → Automates machine learning workflow.
- CountVectorizer → Converts text into numerical features using the Bag of Words (BoW) model.
- SVC → Support Vector Classifier, used for classification tasks.
- RandomForestClassifier → A machine learning classifier that uses multiple decision trees.
- StratifiedKFold → Splits dataset into balanced folds for cross-validation.
- LabelEncoder → Converts categorical labels into numerical format.
- metrics & accuracy_score → Used for evaluating model performance.

Then we read the dataset and count how many times each class appears in the dataset in order to see whether the dataset is balanced or imbalanced. This can be an important preprocessing step because imbalanced datasets may lead to biased models. If one class has significantly more samples, the model may learn to favor that class, while a balanced dataset allows the model to equally learn from all classes.

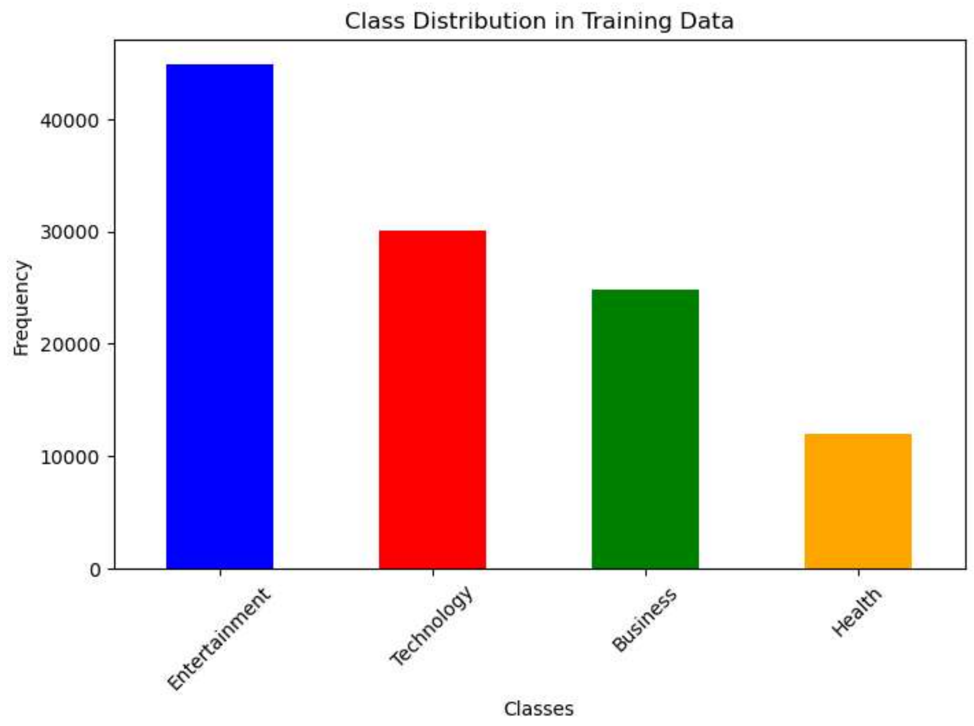


Figure 1.2. *Full Dataset: Entertainment: 44834, Technology: 30107, Business: 24834, Health: 12020.*

We saw that our dataset is imbalanced, so we also created a balanced dataset where each class has the same number of samples. The new sample size for each class is equal to that of the class with the least number of samples, which is 12020 in our case. This balanced dataset will be our second training set. We will compare the two classification methods using both the full and the balanced dataset.

Next, we defined two functions that perform text preprocessing on a dataset. The goal is to clean and standardize the text data before we start the training process. The preprocess_text function takes a single text string as input and applies multiple preprocessing steps to remove noise and standardize the text. If the input text is of null value, the the function returns an empty string (""). Otherwise, the preprocessing steps begin. First, the function converts all characters to lowercase to maintain uniformity. Then, it removes URLs (http\s or www\s), mentions (@usernames), hashtags from text and all non-alphabetic characters (numbers, punctuation and special characters), keeping only letters and spaces. Then, the function uses tokenization to split the text into individual words (tokens) and filters out any token that is not a word (e.g., punctuation or numbers). The function also removes stopwords, which are words that do not carry meaningful information, such as "the", "is", "an", in order to reduce noise. After that, the function converts all words to their base form (Lemmatization), for example "running" becomes "run" and "better" becomes "good". Finally, it joins the cleaned words back into a single string. The final output is a clean, processed text string.

The second function that performs preprocessing is the preprocess_dataset function, which applies the preprocess_text() function mentioned previously to multiple columns in a Pandas DataFrame.

We then applied our text preprocessing functions to our train and test datasets, displaying the progress while processing, merged both text columns ("Title" and "Content") into a single field called "Combined" which helps in text-based machine learning models that expect a single input text field and then saved the cleaned data to CSV files. The final class distribution after preprocessing is shown below.

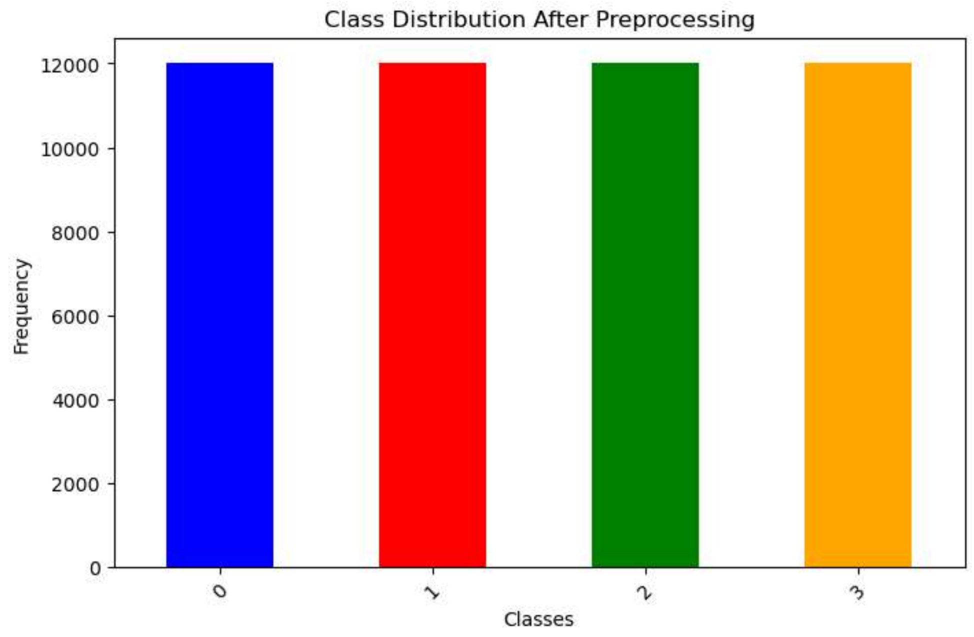


Figure 1.3. *Balanced Training Dataset: Entertainment: 12020, Technology: 12020, Business: 12020, Health: 12020.*

After loading the preprocessed training and test datasets, we started the training process on the entire training dataset. In general, SVM is considered good for text classification with small datasets, while Random Forest is better suited to handle imbalanced, large datasets.

Before we started the training process, we had to perform hyperparameter tuning both for the SVM and the Random Forest classifiers using Bag-of-Words (BoW) feature extraction and GridSearchCV to find the best model configuration for each classifier. GridSearchCV is used to automate hyperparameter tuning by testing different parameter combinations and selecting the best one based on cross-validation performance. For the SVM classifier, we tested both 5.000 and 10.000 number of words in BoW, we included both unigrams (single words) and bigrams (word pairs), and we tested both the Linear and the Radial Basis Function (RBF) kernels. We tested the Regularization parameter as 0.1, 1, 10 and 100 for the Linear kernel and as 10 and 100 for the RBF kernel (lower values allow more margin violations while higher values enforce stricter separation), while for the RBF kernel we tested the gamma classifier parameter both as "auto" and "scale". Linear SVM does not use the gamma classifier because it doesn't involve kernel transformations.

For the Random Forest classifier, we similarly tested both 5.000 and 10.000 number of words in BoW, while also including both unigrams (single words) and bigrams (word pairs). For the Random Forest hyperparameters, the number of trees in the forest was tested as 100 and 200, the maximum depth of trees was tested as none, where the trees grow until all leaves are pure, 10 and 20, which limit the tree's depth for better generalization, the minimum samples required to split a node were tested as 2, which is the default value, and 5, which is more restrictive and reduces overfitting, and the minimum samples per leaf were tested as 1, which is the default value, and 2, which ensures each leaf has at least 2 samples and prevents tiny splits.

We then performed 5-Fold Cross-Validation and trained both models multiple times with different parameter combinations in order to find the best performing one for each method. Cross-Validation is used to prevent overfitting by testing on multiple train-test splits. In our case, it starts by splitting the data into 5 subsets. It then trains the model 5 times, using a different subset for testing each time and computing the accuracy for each fold. In the end, it returns the average accuracy.

Next, we created a pipeline for SVM (Support Vector Machine) with the best parameters found from the Grid Search before. A pipeline chains together multiple steps so that they can be applied sequentially (text processing → model training). The SVM pipeline uses CountVectorizer, which converts text into a matrix where each row represents a document and each column represents a word (with counts of occurrences), in order to extract unigrams (single words) from text (Bag-of-Words model), with the limit set to the 10,000 most common words. The SVM classifier also uses Radial Basis Function (RBF) kernel (kernel = 'rbf'), while the gamma = 'auto' setting sets kernel coefficient automatically. The high value of C (C = 100) means lower regularization (higher complexity), while the class_weight set to 'balanced' adjusts class importance based on frequency.

After that, we created a pipeline for the Random Forest classifier, also with the best parameters found from the Grid Search before. This pipeline also uses CountVectorizer, but in this case it can also extract word pairs alongside single words. The other settings for this pipeline are as follows: n_estimators = 200 means that 200 decision trees are used, while max_depth set to 'None' means that trees grow until all leaves are pure. min_samples_split = 2 means that minimum 2 samples are needed to split, while min_samples_leaf = 1 equals to minimum 1 sample per leaf.

For both methods we also tested the random_state parameter both as None and as a fixed integer, in our case 42. random_state is a seed value that controls the randomness in machine learning models to ensure reproducibility. If set to a fixed integer, the model will produce the same results every time it runs. If set to None, the randomness is not controlled, and the model might yield different results in each run.

The train_and_evaluate_pipeline function was created to perform 5-Fold Cross-Validation. We then trained both the SVM and Random Forest model for both the full and the balanced datasets and for both fixed and random states, and computed the average accuracy for each test case.

SVM Folds	Accuracy
Fold 1	95.14%
Fold 2	94.81%
Fold 3	94.93%
Fold 4	95.15%
Fold 5	95.30%

Table 1.1. *Accuracy of SVM folds during training for the balanced training dataset with a fixed integer as the random_state.*

Random Forests Folds	Accuracy
Fold 1	93.16%
Fold 2	92.17%
Fold 3	93.16%
Fold 4	92.83%
Fold 5	92.90%

Table 1.2. *Accuracy of Random Forests folds during training for the balanced training dataset with a fixed integer as the random_state.*

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**
Accuracy	95.07%	92.84%

Table 1.3. *Accuracy of SVM and Random Forests models for the balanced training dataset with a fixed integer as the random_state.*

For the balanced training dataset with a fixed integer as the random_state, the average accuracy of SVM was 95.07%, while the average accuracy of Random Forest was 92.84%.

SVM Folds	Accuracy
Fold 1	95.09%
Fold 2	95.39%
Fold 3	94.97%
Fold 4	95.37%
Fold 5	94.75%

Table 1.4. *Accuracy of SVM folds during training for the balanced training dataset with "None" as the random_state.*

Random Forests Folds	Accuracy
Fold 1	92.86%
Fold 2	92.79%
Fold 3	92.95%
Fold 4	93.08%
Fold 5	92.52%

Table 1.5. *Accuracy of Random Forests folds during training for the balanced training dataset with "None" as the random_state.*

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**
Accuracy	95.11%	92.84%

Table 1.6. *Accuracy of SVM and Random Forests models for the balanced training dataset with "None" as the random_state.*

For the balanced training dataset with "None" as the random_state, the average accuracy of SVM was 95.11%, while the average accuracy of Random Forest was 92.84%.

SVM Folds	Accuracy
Fold 1	96.154
Fold 2	96.16%
Fold 3	96.14%
Fold 4	96.36%
Fold 5	96.19%

Table 1.7. *Accuracy of SVM folds during training for the full training dataset with a fixed integer as the random_state.*

Random Forests Folds	Accuracy
Fold 1	94.18%
Fold 2	94.09%
Fold 3	94.37%
Fold 4	94.19%
Fold 5	94.09%

Table 1.8. *Accuracy of Random Forests folds during training for the full training dataset with a fixed integer as the random_state.*

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**
Accuracy	96.20%	94.18%

Table 1.9. *Accuracy of SVM and Random Forests models for the full training dataset with a fixed integer as the random_state.*

For the full training dataset with a fixed integer as the random_state, the average accuracy of SVM was 96.20%, while the average accuracy of Random Forest was 94.18%.

SVM Folds	Accuracy
Fold 1	96.31%
Fold 2	96.31%
Fold 3	96.17%
Fold 4	96.06%
Fold 5	96.35%

Table 1.10. *Accuracy of SVM folds during training during training for the full training dataset with "None" as the random_state.*

Random Forests Folds	Accuracy
Fold 1	93.89%
Fold 2	94.21%
Fold 3	94.08%
Fold 4	94.34%
Fold 5	94.35%

Table 1.11. *Accuracy of Random Forests folds during training for the full training dataset with "None" as the random_state.*

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**
Accuracy	96.24%	94.17%

Table 1.12. *Accuracy of SVM and Random Forests models for the full training dataset with "None" as the random_state.*

For the full training dataset with "None" as the random_state, the average accuracy of SVM was 96.24%, while the average accuracy of Random Forest was 94.17%.

We then chose the model with the highest accuracy to make predictions on the test data, which was SVM for the full training dataset with "None" as the random_state in our case. However, due to the randomness, we may actually get better or worse results during multiple test runs, so we ended up excluding all results with the random_state parameter set to "None". The next best result was SVM for the full training dataset with a fixed integer as the random_state, and we saved the output of this test case to testSet_categories.csv.

In conclusion, our chosen training dataset consisted of 111,795 training samples that were distributed between 4 unique classes, while out test dataset consisted of

47,912 unlabeled test samples. Using 5-Fold Cross Validation we saw that the SVM model produced higher average accuracy compared to the Random Forest model, with both models using the Bag-of-Words as a feature extraction method. Comparing tables 1.10 and 1.11, however, we can also see that SVM consistently outperformed Random Forest and there was no case where came close to the accuracy that SVM produced. We can also see that the SVM method consistently outperformed Random Forest in all of our test cases, either with a full or balanced training dataset and with random_state as “None” or a fixed integer. This was because our datasets size is which is not extremely large in the context of deep learning but substantial for traditional machine learning models. SVM tends to perform well on datasets of this size because it focuses on the most important support vectors, reducing the impact of unnecessary noise. Random Forest, on the other hand, works better on very large datasets with millions of instances because it aggregates multiple decision trees, but it struggles to generalize well in high-dimensional sparse spaces.

2. Nearest Neighbor Search with Locality Sensitive Hashing

This task involves the speed up of the K-NN classification (where K=7) method using the LSH technique with the same train and test datasets that we used initially in the text classification task.

For this task we will use and compare the Brute-Force method with Locality Sensitive Hashing. The Brute-Force method is the simplest way to find the K-nearest neighbors (K-NN) of a given query document or data point. It involves comparing every query document with every document in the dataset to find the most similar ones. In our case, each document in the test dataset is compared to each document in the train dataset.

Jaccard Similarity will be used to find the most similar documents. Jaccard Similarity is a measure of how similar two sets are. It is commonly used in text analysis, recommendation systems, and document clustering to compare the similarity between documents, words, or feature sets and is defined as the number of common elements between both sets divided by the total number of unique elements across both sets.

Locality-Sensitive Hashing (LSH) is a technique used to efficiently find similar items in large datasets. It is particularly useful for tasks like near-duplicate detection, document similarity search, and nearest neighbor search. One of the most popular LSH techniques for text and set-based similarity is MinHash LSH, which is based on MinHashing and Jaccard Similarity.

MinHashing is a method that approximates Jaccard Similarity efficiently. Instead of computing the exact Jaccard similarity (which requires set intersection and union), MinHashing creates a compressed representation (signature) of a set while preserving similarity. MinHashing works by converting text/documents into sets of words (shingles or n-grams) and applying multiple random permutations (hash functions) to each set. It then records the minimum hashed value for each permutation and creates a MinHash signature, which is a compact representation of the set. Finally, it compares MinHash signatures instead of full sets to estimate Jaccard Similarity.

The problem with MinHashing is that comparing every pair of signatures is still expensive in large datasets. LSH solves this by reducing the number of comparisons. LSH groups similar MinHash signatures into "buckets" so that only the most likely similar pairs are compared. LSH works by dividing MinHash signatures into bands (groups of hash values) and applying a hash function to each band. If two documents share a bucket in at least one band, they are considered similar candidates. LSH only computes the exact Jaccard Similarity for candidate pairs, significantly reducing the number of comparisons.

MinHash LSH combines MinHashing (to estimate similarity) with LSH (to speed up comparisons). MinHash LSH starts with preprocessing, by computing the MinHash signatures for all documents and storing them in LSH buckets. It then continues with the search phase, where it computes the MinHash for the query document, retrieving candidate documents from LSH buckets and computing the exact Jaccard Similarity only for candidates.

In our code, first we import the required libraries mentioned below:

- pandas → Used for handling datasets (reading CSV, data manipulation).
- re → Used for regular expressions (text processing).
- numpy → Supports numerical computations and array operations.
- nltk → Natural Language Toolkit, used for text processing.
- matplotlib.pyplot → Used for visualizing data.
- sys → Provides system-related functionalities.
- word_tokenize → Splits text into words.
- stopwords → Provides a list of common words to remove (e.g., "the", "is", "and").
- WordNetLemmatizer → Reduces words to their base form (e.g., "running" → "run").
- CountVectorizer → Converts text into numerical features using the Bag of Words (BoW) model.
- time → Measures execution time.
- pairwise_distances → Computes distance between text samples.
- MinHash & MinHashLSH → MinHashing & Locality-Sensitive Hashing (LSH).

Then we read the train and test datasets and we apply to them the same preprocessing functions we created for the previous task, creating a new preprocessed train dataset and a new preprocessed test dataset. After loading the new datasets, we computed the text vectorization using CountVectorizer and saved the new vectorized data for future use. The vectorization of the text data is done by converting text into numeric features using the Bag-of-Words model, allowing only the use of unigrams (single words), using binary encoding (presence or absence of a word) and setting a limit of 5000 words for efficiency. Then the train vectors learns vocabulary from the training data and it converts text to a sparse matrix, while the test vectors uses the learned vocabulary to convert the test data. These vectors are then saved in ".npz" format for faster future loading times. In case precomputed vectorized files already exist (train_vectors.npz, test_vectors.npz), we can either load the precomputed vectors or repeat the process for these files.

After loading the precomputed vectorized data we converted the sparse matrices into dense NumPy arrays using ".astype(bool)", meaning the data is binary (1 if a word appears in a document, 0 if not), which is necessary for Jaccard similarity. The function brute_force_knn_optimized was created to find the k most similar training documents for each test document. The function first computes the Jaccard distance between a test document and all training documents, then it sorts these distances and returns the indices of the closest k neighbors and saves the results to ".npy" files. If these files already exist, we can either load them or recompute them.

Then we implemented Locality-Sensitive Hashing using MinHash to approximate Jaccard similarity for nearest neighbor search in order to compare its performance with the Brute-Force method. We first loaded the precomputed vectorized data in binary form and we created true_knn.npy which stores the true K-nearest neighbors found using the Brute-Force method, and brute_time.npy which stores the time taken for the Brute-Force K-NN search. We also created the create_minhash function which uses MinHash to approximate Jaccard similarity. The function creates different hash functions equal to the number of permutations we give as input to the function, and it retrieves indices of nonzero features in the sparse matrix. If the vector is empty (no features present), it instead inserts a placeholder hash (b'empty') to avoid issues. Otherwise, it hashes each feature index as a string.

The compute_fraction_retrieved function evaluates the LSH performance by comparing LSH-retrieved neighbors with true nearest neighbors. For each test point the function converts true and retrieved neighbors into sets. The function then computes the intersection (common neighbors) and calculates the fraction of true neighbors retrieved (overlap / len(true_set)) before returning the average retrieval fraction across all test samples.

The function lsh_knn_with_fraction was created to initialize the LSH method. It takes as inputs the number of nearest neighbors to retrieve, in this case 7, with the similarity threshold for LSH (higher values mean only very similar vectors are retrieved) equal to 0.2, and number of permutations equal to the number of hash functions. The function then computes the MinHash signatures for each training vector and inserts them into LSH, while also precomputing the MinHashes for the test vectors and calculating the time to do so. It then finds the approximate nearest neighbors for each test vector and with the lsh.query() function retrieves similar items from the index based on the MinHash similarity and records the total query time. Then it measures how many of the true k-nearest neighbors were correctly retrieved by LSH and compares the LSH results with the actual nearest neighbors. Finally, the function returns the time taken to construct the LSH index, the time taken to query the index, the total runtime of the LSH process and the fraction of the true k-nearest neighbors retrieved.

Finally in our code, we ran the LSH method with different numbers of permutations (16, 32, 64), saved the build time, query time, total time, and the retrieval fraction, and compared the results with the Brute-Force method for 4 different thresholds (0.3, 0.5, 0.7, 0.9) and saved the results in the “lsh_brute_force_comparison.csv” file.

Type	Build Time	Query Time	Total Time	Fraction	Permutations	Threshold
Brute-Force-Jaccard	0,00	133483,86831116676	133483,86831116676	100%	-	-
LSH-Jaccard	194,76	0,21	194,97	2,81%	16	0,85
LSH-Jaccard	215,83	0,22	216,05	1,76%	32	0,9
LSH-Jaccard	241,53	0,30	241,83	1,68%	64	0,9

Table 2.1. *Results for threshold = 0,9 (0,85 for 16 permutations due to some error).*

Type	Build Time	Query Time	Total Time	Fraction	Permutations	Threshold
Brute-Force-Jaccard	0,00	133483,86831116676	133483,86831116676	100%	-	-
LSH-Jaccard	208.83	0,32	209,15	4,63%	16	0,7
LSH-Jaccard	225,98	0,44	225,42	4,67%	32	0,7
LSH-Jaccard	314,93	0,72	315,35	4,3%	64	0,7

Table 2.2. *Results for threshold = 0,7.*

Type	Build Time	Query Time	Total Time	Fraction	Permutations	Threshold
Brute-Force-Jaccard	0,00	133483,86831116676	133483,86831116676	100%	-	-
LSH-Jaccard	211,43	1,65	213,08	12,12%	16	0,5
LSH-Jaccard	238,57	1,04	236,62	9,12%	32	0,5
LSH-Jaccard	264,54	1,55	265,69	11,62%	64	0,5

Table 2.3. *Results for threshold = 0,5.*

Type	Build Time	Query Time	Total Timem	Fraction	Permutations	Threshold
Brute-Force-Jaccard	0,00	133483,86831116676	133483,86831116676	100%	-	-
LSH-Jaccard	211,60	22,65	233,25	37,52%	16	0,3
LSH-Jaccard	212,57	33,83	246,4	48,44%	32	0,3
LSH-Jaccard	250,45	5,22	255,68	27,68%	64	0,3

Table 2.4. *Results for threshold = 0,3.*

These results compare Brute-Force Jaccard Similarity Search with Locality-Sensitive Hashing Jaccard Approximate Nearest Neighbor Search under different thresholds and permutations. The error for threshold 0,9 occurs because the MinHashLSH function requires at least 2 bands (b >= 2), and the chosen number of permutations (num_perm = 16) combined with threshold equal to 0.9 results in a calculated number of bands (b) that is too small (b < 2).

We can clearly see that Brute-force search is extremely slow (~133,483 seconds query time) but retrieves 100% of true nearest neighbors, while LSH Jaccard is much faster, but the fraction of true neighbors retrieved depends on the threshold and number of permutations. When the number of permutations is increased from 16 to 32 to 64, we see an increase in build and query times each time and a decrease in the fraction of the true most K documents the LSH method reports, regardless of the threshold we use. The only exception here is for threshold = 0,5, where the fraction is increased when we increase the number of permutations from 32 to 64, which can be considered as an anomaly.

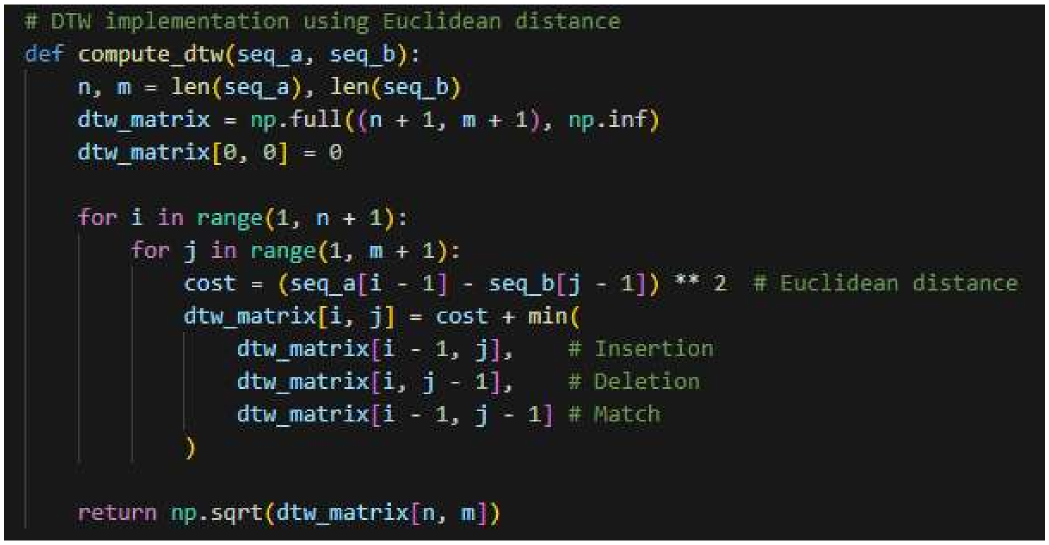
When lowering the threshold from 0,9 to 0,7 to 0,5 to 0,3 we can clearly see fraction and the build and query times increasing each time. The only exception here is again for threshold = 0,5 and 64 permutations, where the build time decreases, which can be considered as an anomaly. In conclusion, Brute-Force is infeasible for large datasets (133,483 seconds is impractical), while LSH is much faster (milliseconds to seconds), but accuracy depends on threshold and permutations. More permutations improve accuracy but increase build time, while lower thresholds return more neighbors but include less similar ones.

3. Time Series Similarity

This task involves the implementation of the algorithm Dynamic Time Warping (DTW) in order to compute the similarities between time series of different time resolutions.

In our code, first we import the required libraries mentioned below:

- pandas → Used for handling datasets (reading CSV, data manipulation).
- numpy → Supports numerical computations and array operations.
- time → Measures execution time.
- sys → Provides system-related functionalities.



```python
# DTW implementation using Euclidean distance
def compute_dtw(seq_a, seq_b):
    n, m = len(seq_a), len(seq_b)
    dtw_matrix = np.full((n + 1, m + 1), np.inf)
    dtw_matrix[0, 0] = 0

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            cost = (seq_a[i - 1] - seq_b[j - 1]) ** 2  # Euclidean distance
            dtw_matrix[i, j] = cost + min(
                dtw_matrix[i - 1, j],    # Insertion
                dtw_matrix[i, j - 1],    # Deletion
                dtw_matrix[i - 1, j - 1] # Match
            )

    return np.sqrt(dtw_matrix[n, m])
```

Figure 3.1. *DTW Implementation using Euclidean distance.*

Figure 3.1 shows our implementation of the Dynamic Time Warping algorithm in the function “compute_dtw”. This function computes the DTW distance between two sequences seq_a and seq_b. The lengths of the two sequences are depicted by n and m. A DTW cost matrix (dtw_matrix) of size (n+1) x (m+1) is initialized with inf (infinity), except for the starting position (0,0), which is set to 0.

The cost function is the squared Euclidean distance between points and is equal to cost = $(seq_a[i-1] - seq_b[j-1])^2$. The DTW cost matrix is updated using either insertion (dtw_matrix[i-1, j]) by skipping an element in seq_a, deletion (dtw_matrix[i, j-1]) by skipping an element in seq_b or matching (dtw_matrix[i-1, j-1]) by aligning both elements. The function returns the final DTW distance which is equal to the square root of the accumulated cost in dtw_matrix[n, m].

In our code, we then proceed to read the “dtw_test.csv” file that contains our time series data, which should have the columns 'series_a' and 'series_b'. We then initialize the variable “results”, which is a list that stores the computed DTW distances, and the variable start_time, which stores the time before the computation starts. Our code then converted the 'series_a' and 'series_b' columns from strings to

NumPy arrays and computed the DTW distance for each pair. Finally, we converted results into a Pandas DataFrame, saved the computed DTW distances to a CSV file (dtw.csv) and created a variable end_time, which stores the time the computation ended.

For our implementation, the time required in order to estimate all the test set can be calculated as: time = end_time - start_time, which in our case is equal to 1426.60 seconds or 23 minutes and 46,6 seconds.