ΤΜΗΜΑ
ΠΛΗΡΟΦΟΡΙΚΗΣ + ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Big Data Mining Techniques (M161) - Winter Semester 2024-2025

Margarita Orfanidi - 7115152400023

Michalis Papageorgiou - 7110132300201

Question 1: Classification Task

Objective

The objective of this task is to classify news articles into four categories: Entertainment, Technology, Business, and Health using two machine learning classification algorithms:

- Support Vector Machines (SVM)
- Random Forests

Data Preprocessing

Text preprocessing is an essential step in Natural Language Processing (NLP) and machine learning. It simplifies raw text data, making it easier for algorithms to process and improving classification performance. The following techniques are basic examples that help transform unstructured text into a structured format, capturing meaningful patterns.

Text normalization ensures consistency by for example converting text to lowercase, removing special characters, and correcting typos. This reduces unnecessary variations and allows algorithms to focus on significant differences.

Tokenization breaks the text into smaller units, such as words or phrases, allowing algorithms to analyze structured data rather than unprocessed text.

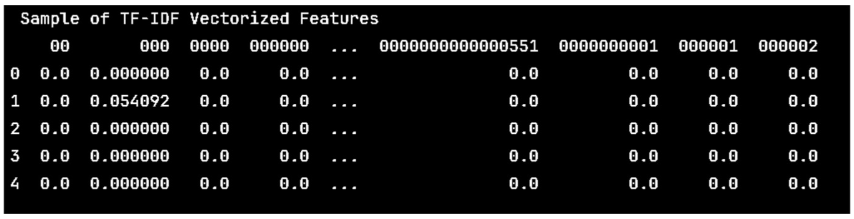
Stopword removal eliminates common words like *the*, *is*, and *and*, which usually don't contribute much to classification accuracy.

Stemming and lemmatization simplify words to their base forms. Stemming reduces words to their root, while lemmatization converts words to their dictionary form (e.g., *studies* → *study*).

Feature engineering transforms text into numerical formats suitable for machine learning algorithms. We applied the Bag of Words (BoW) model with TF-IDF vectorization to represent the text. While BoW captures word frequency, TF-IDF assigns higher importance to rare but meaningful words, reducing the influence of common terms or noisy data and improving classification accuracy by emphasizing key distinguishing features.

To determine the best vectorization approach, we compared CountVectorizer, TfidfVectorizer, and HashingVectorizer. Our experiments showed that **TfidfVectorizer** outperformed the others, making it the most effective method for feature extraction.

Below is an example of how our dataset looks after applying TF-IDF vectorization, highlighting the transformation of text into a numerical format suitable for model training.



```
Sample of TF-IDF Vectorized Features
    00       000  0000  000000  ...  0000000000000551  0000000001  000001  000002
0  0.0  0.000000   0.0     0.0  ...               0.0         0.0     0.0     0.0
1  0.0  0.054092   0.0     0.0  ...               0.0         0.0     0.0     0.0
2  0.0  0.000000   0.0     0.0  ...               0.0         0.0     0.0     0.0
3  0.0  0.000000   0.0     0.0  ...               0.0         0.0     0.0     0.0
4  0.0  0.000000   0.0     0.0  ...               0.0         0.0     0.0     0.0
```

Each row corresponds to a document, and each column represents a unique term. The values indicate TF-IDF scores, most of which are zero, highlighting the sparse nature of the text data. This numerical format is then used as input for our classification models. (Note: The numbers appear on the first line instead of words because numeric values in the text are displayed first.)

Also, for this specific **analysis, a key step is merging the Title and Content fields into a single FullText field**, ensuring that both sources contribute equally to the classification and enhancing feature representation.

Evaluating Preprocessing Techniques

To assess the impact of different preprocessing steps on model performance, we conducted a series of experiments. The goal was to refine the dataset and improve classification accuracy by testing various preprocessing techniques.

The models were evaluated using:

- Linear Support Vector Classifier (LinearSVC) with C=1
- RandomForestClassifier with 100 estimators
- 5-fold cross-validation for robust evaluation

Step 1: Baseline Model - Raw Data Accuracy

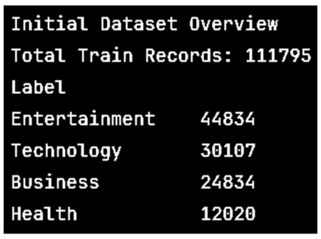
As the first experiment, we train our models using the raw dataset without any preprocessing. This serves as a baseline model for comparison, allowing us to evaluate the effect of subsequent preprocessing steps.

Dataset Statistics (Raw Data)

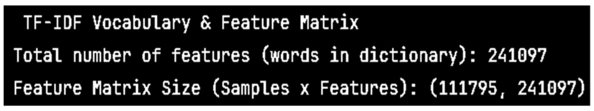
Total Training Records: 111795

Category Distribution:

- Entertainment: 44834
- Technology: 30107
- Business: 24834
- Health: 12020



```
Initial Dataset Overview
Total Train Records: 111795
Label
Entertainment    44834
Technology       30107
Business         24834
Health           12020
```

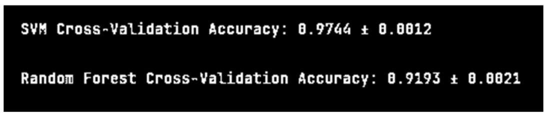


```
 TF-IDF Vocabulary & Feature Matrix
Total number of features (words in dictionary): 241097
Feature Matrix Size (Samples x Features): (111795, 241097)
```

Vocabulary Size: 241097 unique words

Feature Matrix Size (Samples × Features): (111795, 241097)

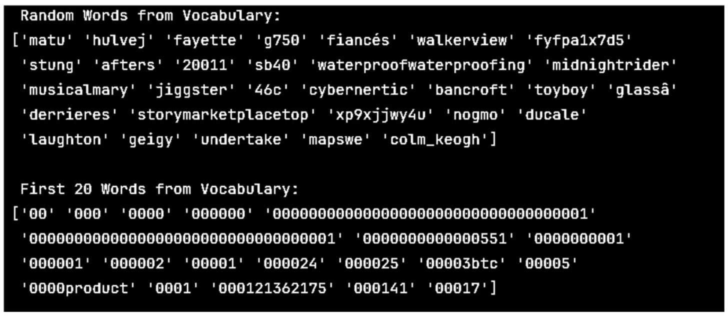
Cross-Validation Results (Raw Data)



```
SVM Cross-Validation Accuracy: 0.9744 ± 0.0012

Random Forest Cross-Validation Accuracy: 0.9193 ± 0.0021
```

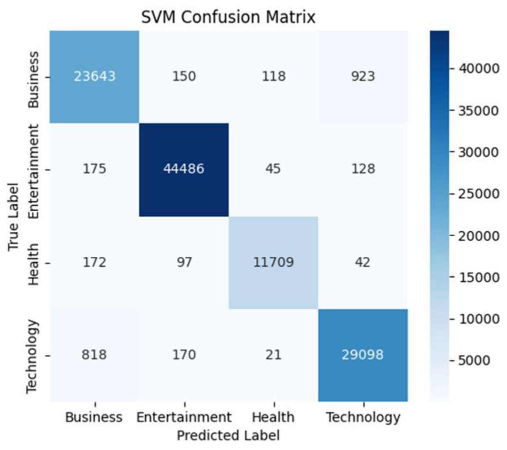
- SVM Mean Accuracy: 97.44% (± 0.12%)
- Random Forest Mean Accuracy: 91.93% (± 0.21%)

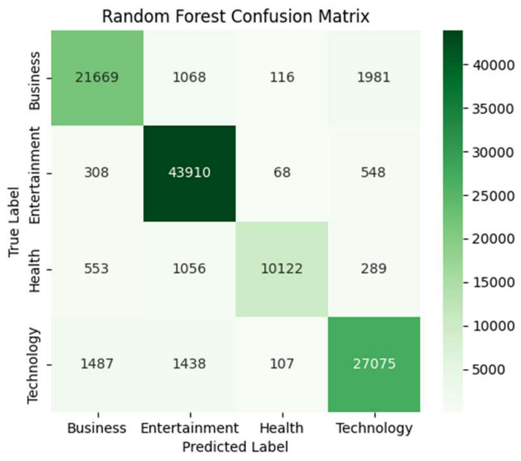
Vocabulary example



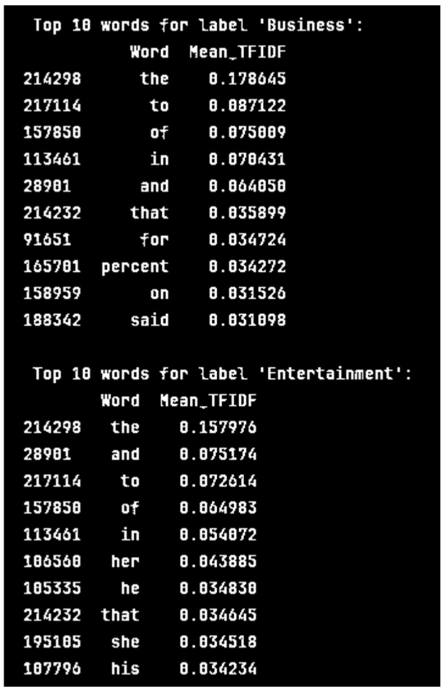
```
 Random Words from Vocabulary:
['matu' 'hulvej' 'fayette' 'g750' 'fiancés' 'walkerview' 'fyfpa1x7d5'
 'stung' 'afters' '20011' 'sb40' 'waterproofwaterproofing' 'midnightrider'
 'musicalmary' 'jiggster' '46c' 'cybernertic' 'bancroft' 'toyboy' 'glassâ'
 'derrieres' 'storymarketplacetop' 'xp9xjjwy4u' 'nogmo' 'ducale'
 'laughton' 'geigy' 'undertake' 'mapswe' 'colm_keogh']

 First 20 Words from Vocabulary:
['00' '000' '0000' '000000' '0000000000000000000000000000000000001'
 '000000000000000000000000000000000001' '0000000000000551' '0000000001'
 '000001' '000002' '00001' '000024' '000025' '00003btc' '00005'
 '0000product' '0001' '000121362175' '000141' '00017']
```

Confusion matrices

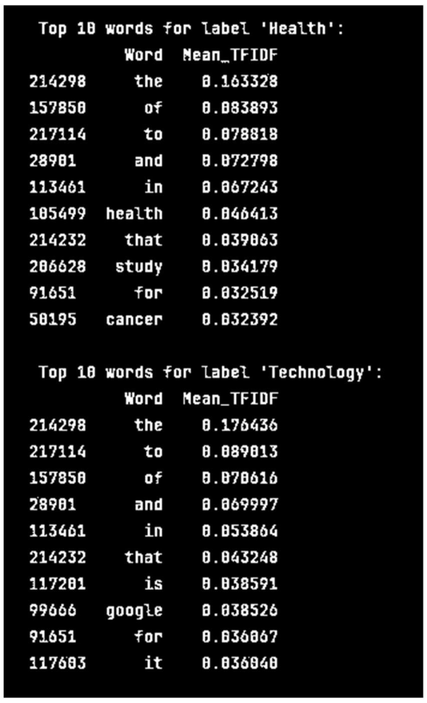




The top 10 words for each class



```
 Top 10 words for label 'Business':
           Word  Mean_TFIDF
214298      the    0.178645
217114       to    0.087122
157850       of    0.075009
113461       in    0.070431
28901       and    0.064050
214232     that    0.035899
91651       for    0.034724
165701  percent    0.034272
158959       on    0.031526
188342     said    0.031098

 Top 10 words for label 'Entertainment':
        Word  Mean_TFIDF
214298   the    0.157976
28901    and    0.075174
217114    to    0.072614
157850    of    0.064983
113461    in    0.054072
106560   her    0.043885
105335    he    0.034830
214232  that    0.034645
195105   she    0.034518
107796   his    0.034234
```



```
 Top 10 words for label 'Health':
          Word  Mean_TFIDF
214298     the    0.163328
157850      of    0.083893
217114      to    0.078818
28901      and    0.072798
113461      in    0.067243
105499  health    0.046413
214232    that    0.039063
206628   study    0.034179
91651      for    0.032519
50195   cancer    0.032392

 Top 10 words for label 'Technology':
          Word  Mean_TFIDF
214298     the    0.176436
217114      to    0.089013
157850      of    0.070616
28901      and    0.069997
113461      in    0.053864
214232    that    0.043248
117201      is    0.038591
99666   google    0.038526
91651      for    0.036067
117603      it    0.036040
```

Observations

The results from our baseline model, trained on raw data without preprocessing, provide valuable insights.

- SVM achieves very high accuracy (97.44%), demonstrating its ability to effectively separate categories even without preprocessing. This is expected, as SVMs perform well in high-dimensional spaces, making them particularly suitable for text classification.
- Random Forest performs slightly worse (91.93%), likely due to the high dimensionality of the feature space. Decision trees, which form the basis of Random Forest, can struggle with very sparse and high-dimensional data.
- The category distribution of the dataset (as shown in the dataset statistics) reveals a significant class imbalance, particularly with Health articles being the least represented category. However, from the confusion matrix, it doesn't seem to be a major issue for model performance. This could be since Health articles contain very specific terminology, which helps the models correctly classify them despite their lower frequency.

- The TF-IDF vocabulary size (241097 words) highlights the large feature space. The feature matrix visualization further demonstrates the sparsity of the text representation, where most TF-IDF values are zero.
- Based on the confusion matrices, the SVM model shows clearer separation between categories and commits fewer overall misclassifications than the Random Forest model. SVM performs strongly across all classes, though some confusion exists specifically between Business and Technology articles. In contrast, the Random Forest model struggles across multiple categories, with notable misclassifications between Business and Technology and Entertainment and Technology. This indicates that Random Forest has difficulty distinguishing overlapping features among various categories, making SVM the more reliable model for this classification task.
- The TF-IDF analysis of the top words per category reveals that articles share common high-frequency words, which emphasizes the importance of stopword removal and improved feature selection for better classification accuracy. Even in the presence of stopwords or noise data for example numbers and special characters, the TF-IDF method assigns high importance to meaningful words like "cancer" and "health" for the Health category, as they are distinctive and carry significant relevance for that particular category, helping the model distinguish it from others.
- The vocabulary extracted from the dataset highlights key insights into the text representation and sparsity of features:

    - The first section of the vocabulary displays a set of 20 randomly selected words, including meaningful terms ("cybernetic," "bancroft") and non-standard words ("xpXpjxjwy4u"). This suggests that some preprocessing techniques (such as stemming, lemmatization, and typo correction) may help refine the dataset and remove unnecessary noise.
    - The second section shows the first 20 words from the vocabulary, which includes numerical sequences, special characters, and tokens that may not contribute meaningfully to classification. These are likely artifacts from data collection or encoding and could be removed through preprocessing steps like regex filtering.
    - The presence of highly specific and uncommon words emphasizes the need for feature selection techniques to prevent overfitting and improve generalization in the classification models.

These initial results set a benchmark for comparison as we proceed with text preprocessing techniques to further refine our classification models.

Step 2: Removing Duplicate Entries

To ensure data consistency and remove redundant information, different types of duplicate removal were applied:

1. ID-based duplicate removal: Since each article has a unique identifier, no records were eliminated.

2. Title-based duplicate removal: Articles with identical titles were removed.

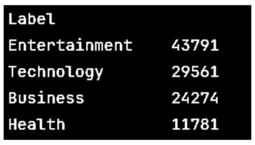
3. Content-based duplicate removal: Further eliminated articles with identical content, ensuring only unique articles remain.

Dataset Statistics (After Duplicate Removal)

- Total Records After ID-based Removal: 111795
- Total Records After Title-based Removal: 110634
- Total Records After Content-based Removal: 109407

Category Distribution:

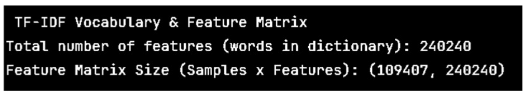
  - Entertainment: 43791
  - Technology: 29561
  - Business: 24274
  - Health: 11781



```
Label
Entertainment    43791
Technology       29561
Business         24274
Health           11781
```

Vocabulary Size: 240552 unique words

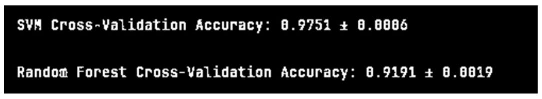
Feature Matrix Size: (109407, 240552)



```
 TF-IDF Vocabulary & Feature Matrix
Total number of features (words in dictionary): 240240
Feature Matrix Size (Samples x Features): (109407, 240240)
```

Cross-Validation Results (After Duplicate Removal)

- SVM Mean Accuracy: 97.51% (± 0.06%)
- Random Forest Mean Accuracy: 91.91% (± 0.19%)



```
SVM Cross-Validation Accuracy: 0.9751 ± 0.0006

Random Forest Cross-Validation Accuracy: 0.9191 ± 0.0019
```

Observations

- SVM accuracy improves slightly from 97.44% (± 0.12%) to 97.51% (± 0.06%) after duplicate removal, indicating that eliminating redundant entries reduces noise and helps the model generalize better.
- Random Forest accuracy remains stable at 91.91% (± 0.21% → ± 0.19%), suggesting that duplicate removal has minimal impact on performance.
- Efficiency gains are evident as duplicate removal reduces computational complexity without negatively affecting model accuracy, allowing for faster training and predictions on large datasets.
- Vocabulary size decreases slightly from 241097 to 240552 unique words, indicating that some duplicates at the title differ a bit in content.

We will retain this enhancement in the following steps for both models.

Step 3: Stopword Removal

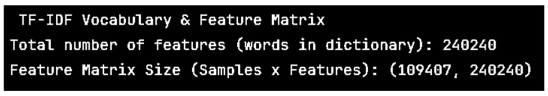
As observed in the important words for each class in Step 1, our texts contain many frequently repeated words like “the,” “for,” “is,” “to,” etc. These words appear often but offer little semantic value for classification.

First experiment

First, we tested removing only the default stopwords provided by Scikit-learn's .
The results were as follows:

Dataset Statistics (After Stopword Removal)

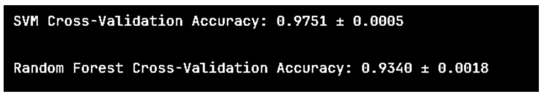
- Vocabulary Size: 240240
- Feature Matrix Size (Samples × Features): (109407, 240240)



```
 TF-IDF Vocabulary & Feature Matrix
Total number of features (words in dictionary): 240240
Feature Matrix Size (Samples x Features): (109407, 240240)
```

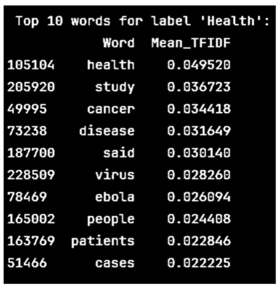
Cross-Validation Results (After Stopword Removal)

- SVM Mean Accuracy: 97.51% (± 0.05%)
- Random Forest Mean Accuracy: 93.40% (± 0.18%)

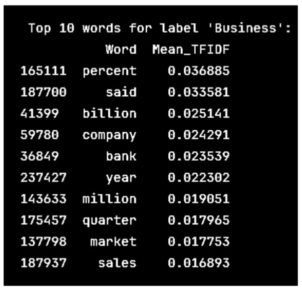


```
SVM Cross-Validation Accuracy: 0.9751 ± 0.0005

Random Forest Cross-Validation Accuracy: 0.9340 ± 0.0018
```

Also some examples of how the top words changes after this removal.



```
 Top 10 words for label 'Health':
            Word  Mean_TFIDF
105104    health    0.049520
205920     study    0.036723
49995     cancer    0.034418
73238    disease    0.031649
187700      said    0.030140
228509     virus    0.028260
78469      ebola    0.026094
165002    people    0.024408
163769  patients    0.022846
51466      cases    0.022225
```



```
 Top 10 words for label 'Business':
           Word  Mean_TFIDF
165111  percent    0.036885
187700     said    0.033581
41399   billion    0.025141
59780   company    0.024291
36849      bank    0.023539
237427     year    0.022302
143633  million    0.019051
175457  quarter    0.017965
137798   market    0.017753
187937    sales    0.016893
```

Observations

- SVM accuracy remains stable at 97.51% after stopword removal, similar to the result after duplicate removal alone.
- Random Forest Accuracy improves from 91.91% to 93.40%, marking a notable gain of around 1.5%. Removing irrelevant features (stopwords) allows the model to focus on more meaningful splits.
- While stopword removal has no measurable effect on SVM performance, it significantly enhances the accuracy and efficiency of Random Forest. This highlights the importance of tailored preprocessing based on the algorithm’s sensitivity to irrelevant features.
- Words like “said” could still be considered for removal to further improve performance.
- Computational efficiency improves for both models due to the reduced feature space size, from 240552 to 240240 words. This reduction decreases training time, particularly

benefiting the Random Forest model, which is more sensitive to high-dimensional and sparse data.

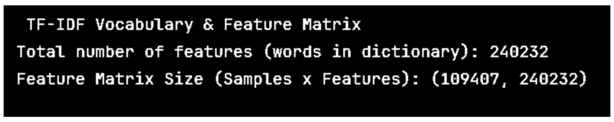
Second experiment

In the second experiment, additional custom stopwords were removed from the dataset to further refine the feature space and reduce noise. These custom words were selected based on their high frequency across all categories but minimal contribution to classification accuracy.

The following custom stopwords were removed:

“said", "say", "year", "time", "just", "make", "like", "also", "last", "would", "could", "may", "new", "one", "first"

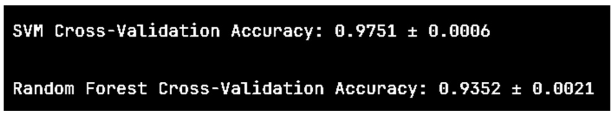
Dataset Statistics (After Custom Stopword Removal)



```
 TF-IDF Vocabulary & Feature Matrix
Total number of features (words in dictionary): 240232
Feature Matrix Size (Samples x Features): (109407, 240232)
```

- Vocabulary Size: 240232
- Feature Matrix Size (Samples × Features): (109407,240232)

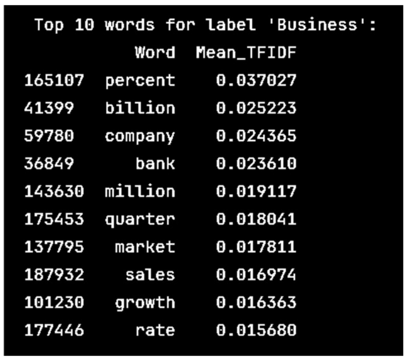
Cross-Validation Results (After Custom Stopword Removal)



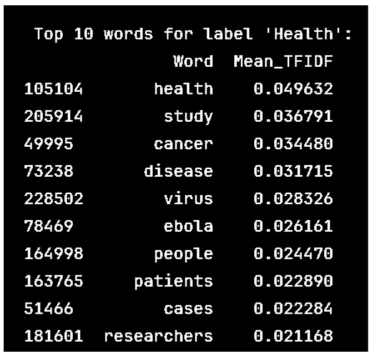
```
SVM Cross-Validation Accuracy: 0.9751 ± 0.0006

Random Forest Cross-Validation Accuracy: 0.9352 ± 0.0021
```

- SVM Accuracy: 97.51% (± 0.06%)
- Random Forest Accuracy: 93.52% (± 0.21%)

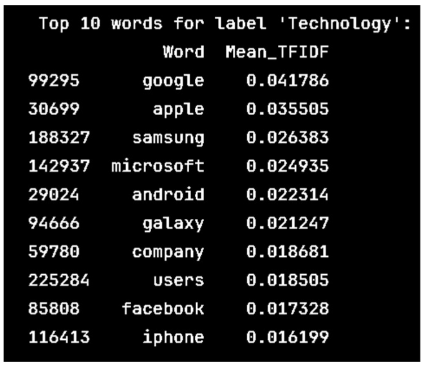
Top Words



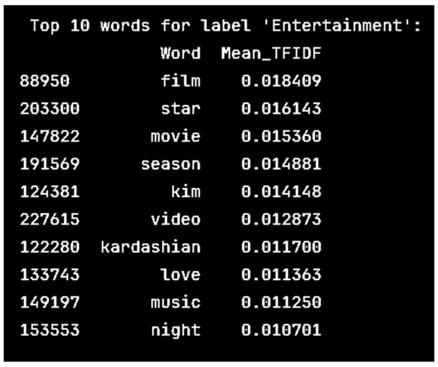
```
 Top 10 words for label 'Business':
          Word  Mean_TFIDF
165107  percent    0.037027
41399   billion    0.025223
59780   company    0.024365
36849      bank    0.023610
143630  million    0.019117
175453  quarter    0.018041
137795   market    0.017811
187932    sales    0.016974
101230   growth    0.016363
177446     rate    0.015680
```



```
 Top 10 words for label 'Health':
                Word  Mean_TFIDF
105104        health    0.049632
205914         study    0.036791
49995         cancer    0.034480
73238        disease    0.031715
228502         virus    0.028326
78469          ebola    0.026161
164998        people    0.024470
163765      patients    0.022890
51466          cases    0.022284
181601   researchers    0.021168
```



```
 Top 10 words for label 'Technology':
              Word  Mean_TFIDF
99295       google    0.041786
30699        apple    0.035505
188327     samsung    0.026383
142937   microsoft    0.024935
29024      android    0.022314
94666       galaxy    0.021247
59780      company    0.018681
225284       users    0.018505
85808     facebook    0.017328
116413      iphone    0.016199
```



```
 Top 10 words for label 'Entertainment':
                Word  Mean_TFIDF
88950           film    0.018409
203300          star    0.016143
147822         movie    0.015360
191569        season    0.014881
124381           kim    0.014148
227615         video    0.012873
122280    kardashian    0.011700
133743          love    0.011363
149197         music    0.011250
153553         night    0.010701
```

Observations

- The SVM accuracy remains **stable** at 97.51% (± 0.06%) even after applying custom stopword removal.
- The Random Forest accuracy shows a slight improvement, increasing to 93.52% (± 0.21%) from the previous 93.41%. This suggests that Random Forest benefits from a cleaner feature space, allowing it to make more accurate splits during classification.
- The vocabulary size decreases to 240,232 after custom stopword removal. This reduction improves computational efficiency by narrowing the feature space and eliminating redundant information, making the model faster and more efficient.
- Category-specific terms become clearer: For example in Technology, words like "google", "apple", and "android" dominate, reflecting industry-relevant terms, in Business, financial terms such as "percent", "company", and "rate" become more prominent.

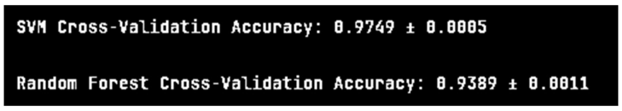
We will retain this enhancement in the following steps for both models.

Step 4: Feature Selection – Limiting Maximum Features and balancing classes

First experiment

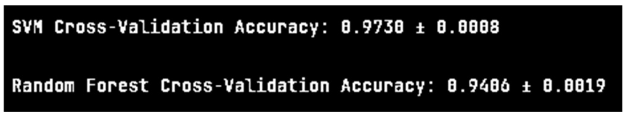
To further optimize computational efficiency and reduce memory usage, feature selection was applied by limiting the vocabulary size to various thresholds. This approach ensures that only the most informative and relevant words are retained for classification, while less significant terms are discarded. By inspecting random words from the dataset, it became evident that many terms were irrelevant or contributed little to the classification task, reinforcing the need for this filtering process.

Vocabulary Size : 50000



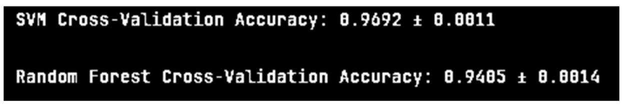
```
SVM Cross-Validation Accuracy: 0.9749 ± 0.0005

Random Forest Cross-Validation Accuracy: 0.9389 ± 0.0011
```

Vocabulary Size : 20000



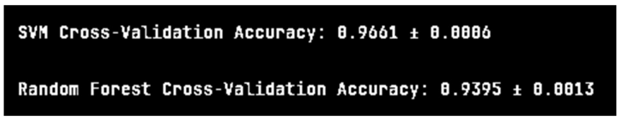
```
SVM Cross-Validation Accuracy: 0.9730 ± 0.0008

Random Forest Cross-Validation Accuracy: 0.9406 ± 0.0019
```

Vocabulary Size : 10000



```
SVM Cross-Validation Accuracy: 0.9692 ± 0.0011

Random Forest Cross-Validation Accuracy: 0.9405 ± 0.0014
```

Vocabulary Size : 7000



```
SVM Cross-Validation Accuracy: 0.9661 ± 0.0006

Random Forest Cross-Validation Accuracy: 0.9395 ± 0.0013
```

Observations

- The accuracy of SVM decreases as the number of features is reduced, indicating that it performs better with a larger feature space. On the other hand, Random Forest achieves its best accuracy when the number of features is limited, suggesting that reducing the feature set helps prevent overfitting and improves predictions. The optimal performance is seen with 20,000 features with accuracy 94,86%.

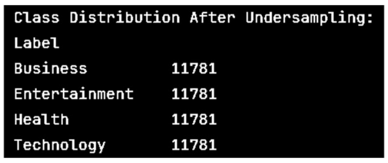
Second Experiment

A significant imbalance was observed in the dataset, with some categories containing substantially more samples than others. The Entertainment and Technology categories had a larger number of articles compared to Business and Health.

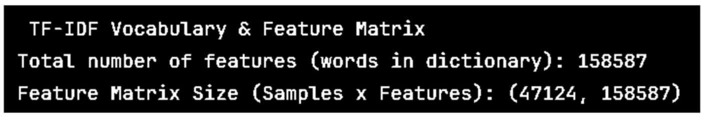
To mitigate this issue, undersampling was applied. This means that the number of articles in each category was reduced to match the class with the lowest sample count. By doing this, we ensure that each category is equally represented, avoiding bias towards the dominant classes. After applying undersampling, each category contains exactly 11781 records, creating a balanced dataset.

Dataset Statistics (After Undersampling)

- Class Distribution After Undersampling:
  - Business: 11781
  - Entertainment: 11781
  - Health: 11781
  - Technology: 11781



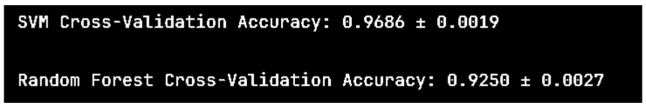
```
Class Distribution After Undersampling:
Label
Business         11781
Entertainment    11781
Health           11781
Technology       11781
```



```
 TF-IDF Vocabulary & Feature Matrix
Total number of features (words in dictionary): 158587
Feature Matrix Size (Samples x Features): (47124, 158587)
```

- Vocabulary Size: 158587 unique words
- Feature Matrix Size: (47124, 158587)

Cross-Validation Results (After Undersampling)



```
SVM Cross-Validation Accuracy: 0.9686 ± 0.0019

Random Forest Cross-Validation Accuracy: 0.9250 ± 0.0027
```

- SVM Accuracy: 96.86% (± 0.19%)
- Random Forest Accuracy: 92.50% (± 0.27%)

Observations

Both models experienced a slight drop in accuracy after undersampling.

- SVM accuracy (97.51% → 96.86%).
- Random Forest accuracy (93.52% → 92.50%).

Reducing the dataset size removes potentially useful training information, which can negatively impact classification performance. Although undersampling ensures fair class representation, it comes at the cost of losing valuable data that might have been beneficial—especially for categories with high linguistic variability.

This approach is not suitable for either model

Step 5: Cleaning and Lemmatization

To improve text consistency and eliminate unwanted noise, we applied several preprocessing steps to standardize the dataset.

Converting text to lowercase, ensuring uniform representation across all documents. This part its happening in all steps by default using **TfidfVectorizer.**

First experiment

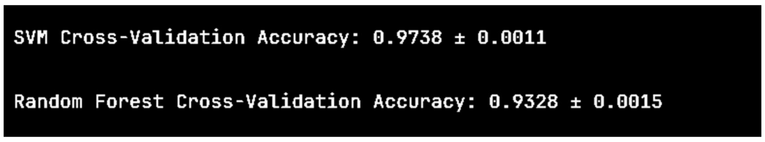
Removing special characters, numbers, and extra spaces, retaining only meaningful words.

The function, clean_text, performs basic text preprocessing by:

Removing special characters and numbers:
The regular expression re.sub(r"[^a-zA-Z\s]", "", text) removes all characters that are **not** alphabetic (A-Z, a-z) or whitespace (\s).

Normalizing spaces:
The regular expression re.sub(r"\s+", " ", text).strip() replaces multiple consecutive spaces with a single space and removes any leading or trailing spaces.

The results of the performance

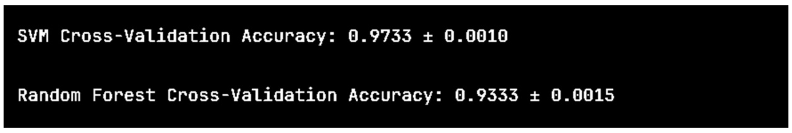


```
SVM Cross-Validation Accuracy: 0.9738 ± 0.0011

Random Forest Cross-Validation Accuracy: 0.9328 ± 0.0015
```

Second experiment

Additionally to the previous experiment we apply aplso lemmatization, converting words to their base form (e.g., "running" → "run", "better" → "good"). This process helps to reduce vocabulary size while preserving the core meaning of words, improving classification efficiency. For the lemmatization process, the **Natural Language Toolkit (NLTK)** library was utilized. Specifically, the WordNetLemmatizer class from the nltk.stem module was employed to reduce words to their base form, known as the lemma.

The results of the performance


```
SVM Cross-Validation Accuracy: 0.9733 ± 0.0010

Random Forest Cross-Validation Accuracy: 0.9333 ± 0.0015
```

Observations

- After applying both text cleaning (removing special characters, numbers, and normalizing spaces) and lemmatization (converting words to their base forms), both models exhibited a slight decrease in accuracy compared to earlier results.

Given this decline in performance, lemmatization and text cleaning will not be included as preprocessing steps in the final data preparation pipeline.

Evaluating Classification Models

To refine the optimal settings for each classifier, a series of experiments were conducted to evaluate how different parameter configurations affect both accuracy and training time.

Random Forest Classifier

The Random Forest algorithm is a robust and efficient classification technique, particularly effective in handling high-dimensional data. It is well-suited for text classification due to its ability to manage non-linear relationships and its high interpretability. The model was tested using various numbers of estimators to determine the best balance between performance and training time.

Results:

- n_estimators = 100
  - Accuracy: 93.85%
  - Training Time: 63.61 seconds
- n_estimators = 500
  - Accuracy: 94.14%
  - Training Time: 321.19 seconds
- n_estimators = 800
  - Accuracy: 94.17%
  - Training Time: 517.77 seconds
- n_estimators = 1500
  - Accuracy: 94.12%
  - Training Time: 986.75 seconds

- n_estimators = 3000
    - Accuracy: 94.15%
    - Training Time: 2017.85 seconds

Observations:

- Increasing n_estimators beyond 1500 did not improve the accuracy.
- The best trade-off between accuracy and efficiency was observed with 800 estimators, achieving an accuracy of 94.17% with a reasonable training time.

Support Vector Machine (SVM)

Support Vector Machines (SVMs) are particularly effective for text classification, as they handle high-dimensional data efficiently. SVMs work by finding an optimal hyperplane that maximizes the margin between categories, making them well-suited for linearly separable data. To improve computational efficiency with large datasets, the **LinearSVC** implementation was used.

The model was trained using different values for the regularization parameter C, which controls the trade-off between maximizing the margin and minimizing classification errors.

Results:

- **C = 0.1** → Accuracy: 96.80%
- **C = 1** → Accuracy: 97.45%
- **C = 10** → Accuracy: 97.08%
- **C = 100** → Accuracy: 96.78%

Observations:

- The best performance was achieved with C = 1, resulting in an accuracy of 97.45%.
- Increasing C beyond this point led to overfitting, causing a slight decrease in accuracy.

Conclusion

Based on the experimental results, the final models were configured with the following parameters:

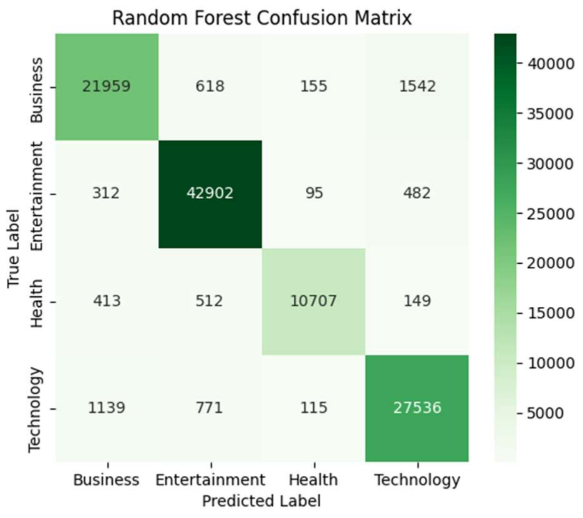
Model Parameters:

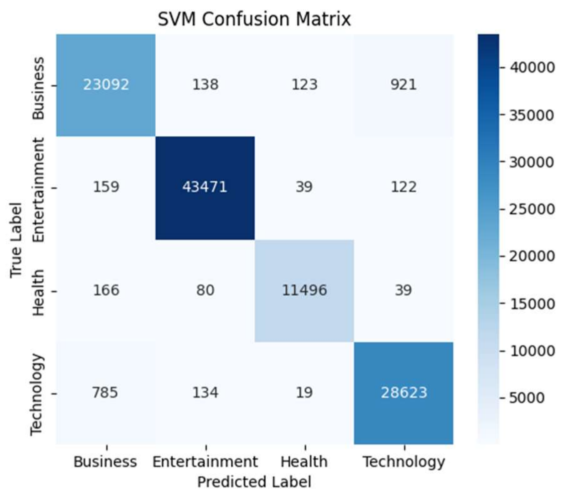
- Support Vector Machine: Regularization parameter (C) set to 1 to balance generalization and classification performance.
- Random Forest: Number of estimators (n_estimators) set to 800, providing the best trade-off between accuracy and training time.

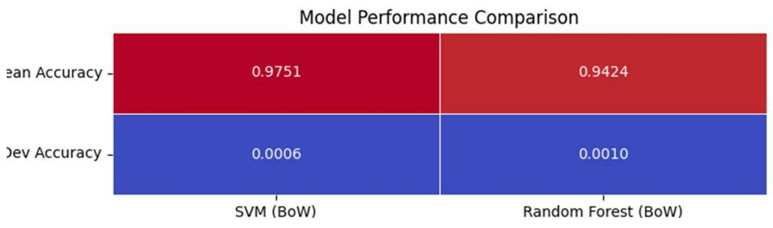
Preprocessing Strategy:

- SVM: Applied stopword removal and duplicate record elimination to maintain a rich feature space and preserve valuable information.
- Random Forest: Applied stopword removal, duplicate record elimination, and feature selection by limiting the vocabulary size to 20,000 words.

Results of Final Models







Confusion Matrix Analysis:

- **SVM Model:**
    - Demonstrated superior classification performance across all categories.
    - Achieved high precision for categories like Entertainment and Technology, with minimal misclassifications.
    - For instance, Entertainment articles were correctly classified in 48,471 cases, with very few errors.

    - Minor misclassifications occurred between closely related categories (e.g., Business and Technology), indicating that the SVM model effectively distinguished between similar topics.
- **Random Forest Model:**
    - While the Entertainment category achieved 42,002 correct classifications, the model faced challenges distinguishing between Business and Technology.
    - The algorithm exhibited difficulty managing sparse datasets, resulting in increased false positives for categories like Business and Technology compared to SVM.

Performance Metrics:

- **SVM:**
    - Achieved a higher mean accuracy of 97.51%.
    - Showed a lower standard deviation (σ) of 0.0006, indicating consistent performance across different cross-validation folds.
- **Random Forest:**
    - Reached a mean accuracy of 94.24%.
    - Displayed a higher standard deviation (σ) of 0.0010, suggesting greater variability across different training runs.

Final Decision:

The SVM model outperformed the Random Forest classifier in terms of both accuracy and consistency. Given its robustness, ability to handle high-dimensional data effectively, and superior generalization capabilities, **SVM** was selected as the final model for the classification task.

This choice ensures:

- **Optimal performance** across all categories.
- **Consistency** in classification results.
- **Efficient handling** of large, high-dimensional text datasets generated using **TF-IDF vectorization**.

The Random Forest model, while effective, demonstrated limitations in handling sparse data and required more extensive preprocessing to reach near-optimal results. Thus, SVM offers a more reliable solution for the text classification problem presented in this project.

The final code for the models is contained in the file FinalModels.py. The code for experiments related to model parameter evaluation is in the files RandomForestEvaluation.py and SVMevaluation.py. Additionally, the file EvaluationDataPreprocessingTechniques.py contains the experiments for the data processing, with some parts commented out. The file testSet_categories.csv contains the saved results.

Question 2: Nearest Neighbor Search without and with Locality Sensitive Hashing

Objective

The purpose of the exercise is to find the nearest neighbors using the Brute Force Jaccard method and Locality Sensitive Hashing, as well as to compare their results.

Implementation algorithms

In two sets, the Jaccard index is equal to the fraction of the intersection over their union, and it indicates their similarity. The Brute Force Jaccard method examines all elements in order to find the closest ones. In this particular case, every row of the file «test_without_labels» was examined with every row of the file «train» in order to find the nearest neighbors. The Brute Force Jaccard algorithm always finds the nearest neighbors but is quite slow and is not recommended for large volumes of data.

The Locality Sensitive Hashing algorithm, in combination with the Minhash technique, is used to find similar objects between two sets. The Minhash technique is used to create "signatures" for the dataset. The LSH algorithm groups sets with similar signatures. The Minhash–LSH technique is not as effective as the Brute Force Jaccard, but it is much faster.

Method of Implementation

As expected, the major challenge in implementing the code that compares the results and execution time of the two algorithms lies mainly in the implementation of the Brute Force Jaccard. This is due to the volume of data in the two files.

Finding the intersection and union for each element of each file and computing the Jaccard fraction is a very time-consuming process that would require a considerable amount of implementation time. Also, the attempt to implement using the nearestNeighbors method failed when the computation was done for more than 10% of the elements due to insufficient RAM (over 100GB of RAM was required).

To avoid memory errors and optimize speed in the Brute Force method, techniques based on Sparse Matrices and Chunk Processing were used. The texts were converted into shingles using CountVectorizer, which were represented as Sparse Matrices (CSR format) to avoid storing full matrices in memory. The processing of the documents was divided into chunks of 1000 documents, in order to limit simultaneous data loading and control memory consumption.

For the calculation of Jaccard similarity, Vectorized Operations with Matrix Multiplication (T.dot(R.T)) were applied for computing intersections, and Precomputed Row Sums for unions, thus avoiding loops. Finally, the use of a Min-Heap with negative distance allows for the efficient selection of the nearest neighbors, further reducing the computation time.

The implementation of the MinHash LSH method is based on specific functions from the datasketch library for creating and managing MinHash signatures. Specifically, for each document, the set of shingles (with a size of 6 characters) is first created, and then, using the MinHash function from datasketch, its signature is generated. This signature constitutes a compact representation of the original set of shingles, allowing for fast comparisons without the need for complete processing of all the elements of the text.

After creating the MinHash signatures, the code utilizes the MinHash LSH class from the datasketch library to organize the documents into buckets based on the defined threshold. Specifically, each document is inserted into the LSH index using the insert() method, where the document's ID and the corresponding MinHash signature are provided. During the query, the query() method is called for each test document, thereby retrieving the candidate neighbors that are located in the same bucket.

Data Preprocessing

The preprocessing applied to the files begins with loading the two CSVs into a DataFrame. Next, for each document, the "Title" and "Content" columns are merged into a new column (e.g., "FullText"), where spaces are inserted between the two fields to maintain the smooth flow of the text. During the merging process, missing values (NaN) are replaced with empty strings to avoid errors. After that, processing is applied to standardize the text: all characters are converted to lowercase, extra spaces are eliminated, and any leading or trailing spaces are removed using the appropriate method. Finally, to reduce the size of the DataFrame, the original "Title" and "Content" columns are removed, retaining only the processed column.

Preprocessing is critical for the subsequent phases of the analysis, as clean and uniform text ensures that the similarity analysis is based on comparable and properly formatted data, thereby reducing errors and memory consumption.

Shingle Size

To determine the optimal shingle size k, the length of the documents and the diversity of characters are taken into account. Shingle size must be large enough to minimize the likelihood of random shingle overlap among different documents. In the specific datasets, the average is 429 words per document, which corresponds to approximately 2150 characters (assuming 5 characters per word). This size places the documents in an intermediate category—larger than an email but smaller than an academic article.

The choice of k=6 emerges as optimal for this case. According to the empirical rule of 27^k (27 possible characters per position), 27^6 yields 387420489 possible shingles—a number far greater than the average document length. This ensures that the probability of random shingle overlap in unrelated documents remains minimal. Additionally, k=6 addresses the problem of frequent characters that reduce the effective diversity of shingles, thus avoiding false positive similarities in the Jaccard index.

In comparative analysis, for short texts (such as emails) k=5 is recommended, while for long texts (e.g., academic articles) k=9 is suggested. However, for medium-sized documents, a shingle length of 6 offers the optimal balance: it minimizes both random shingle overlap and computational load. This strategy ensures that even documents with repeated characters will have significantly distinct shingle sets, improving the accuracy of similarity detection.

Parameters for the implementation of Min Hash – Locality Sensitive Hashing

Threshold: The threshold in LSH determines the minimum level of similarity for two objects to be considered as "near neighbors". This limit is implicitly defined by parameters such as the number of bands (b) and the number of rows (r) in MinHash, according to the formula: $(1/b)^{\wedge}(1/r)$. When the threshold is high, precision increases but false negatives also increase. Conversely, a lower threshold allows the examination of more pairs, though with the potential inclusion of more false positives.

Permutations: In the MinHash method, permutations refer to random rearrangements of the elements of a set, which determine the order in which the data appears. Each permutation "indirectly selects" the first element (with the smallest hash value) that will be included in the signature of the set. The more permutations that are used, the more reliable the estimation of the Jaccard similarity between documents becomes, as a wider range of possible arrangements is covered. However, computational resources also increase. The number of permutations plays a critical role in the sensitivity of the method, as fewer permutations reduce the accuracy of the method while more permutations apply stricter classification criteria that lead to greater accuracy.

Evaluating Different Parameters

For measuring the performance and execution time of the MinHash-LSH algorithm in comparison with the Brute Force Jaccard, tests were conducted with different parameters:

- Multiple Threshold values were tested, which affect how "close" objects are considered.
- The number of Permutations in MinHash was varied, which determine the accuracy of the Jaccard similarity estimation.
- Various neighbor counts were examined to measure the effect of the problem size on performance.

Finally, since the MinHash LSH class from the datasketch library returns all the objects that are in the same "bucket" and from these we randomly take a number equal to the number of nearest neighbors, the application of the Brute Force Jaccard method was tested among the objects in the same bucket and the selection of the closest objects.

Permutations Test

Initially, tests were conducted with different permutations, more specifically for 16, 32, and 64. The threshold chosen is 0.8. It is noted that for a number of permutations equal to 16, the threshold could not be 0.9, and therefore 0.8 was chosen. The table below shows the execution times (Build Time, Query Time, and Total Time), the percentage of common similar files between the two methods, and finally the parameters on which the code was executed.

Type	Build Time (s)	Query Time (s)	Total Time (s)	Fraction	Parameters
B.F. - Jaccard	0.00	9050.31	9050.31	100%	K = 7
LSH - Jaccard	2981.82	2.46	2984.28	2%	Perm = 16, Thres = 0.8, K = 7
LSH - Jaccard	3011.67	2.60	3014.27	2%	Perm = 32, Thres = 0.8, K = 7
LSH - Jaccard	3401.10	2.72	3408.82	2%	Perm = 64, Thres = 0.8, K = 7

From the results of the table, the LSH-Jaccard method exhibits a total execution time of approximately 3000 seconds for all permutation values (16, 32, 64), meaning it is 3 times faster than the Brute Force Jaccard (9050 seconds). The Query Time for LSH remains extremely low (around 2.5 seconds), as the algorithm restricts candidate neighbors through the LSH buckets. However, the critical issue concerns the percentage of common similar files between the two methods, which remains at 2% for all LSH settings.

This low percentage indicates that LSH, with a threshold of 0.8, identifies only 2% of the pairs considered similar by the Brute Force approach. In other words, LSH misses 98% of the actual similar files, even though it is significantly faster. The cause of this large discrepancy is attributed to the high threshold: the higher the similarity threshold, the fewer pairs are included in the LSH buckets.

Despite increasing the number of permutations from 16 to 64, the Fraction remains at 2%, which further suggests that the choice of threshold is the main parameter determining LSH's performance. The permutations mainly affect the Build Time, which increases from 2981.82 to 3408.82 seconds, due to the greater computational complexity.

Threshold Test

Since the first test was conducted with different permutation values, we now test different threshold values, specifically 0.2, 0.5, and 0.8. The table below shows the execution times (Build Time, Query Time, and Total Time), the percentage of common similar files between the two methods, and finally the parameters on which the code was executed.

Type	Build Time (s)	Query Time (s)	Total Time (s)	Fraction	Parameters
B.F. - Jaccard	0.00	9374.22	9374.22	100%	K = 7
LSH - Jaccard	3489.06	3.01	3492.07	2%	Perm = 64, Thres = 0.8, K = 7
LSH - Jaccard	3497.85	3.92	3501.77	6%	Perm = 64, Thres = 0.5, K = 7
LSH - Jaccard	3522.63	21.56	3544.20	0%	Perm = 64, Thres = 0.2, K = 7

From the table, it appears that lowering the threshold leads to an increase in Query Time while also affecting the percentage of common similar files.

For a threshold of 0.8, the results are consistent with previous findings, as the Total Time remains low (~3492 sec) and the Fraction is 2%, confirming that a high threshold drastically limits the results to high-similarity pairs. With a threshold of 0.5, the Fraction increases to 6%, while the Query Time is only slightly higher (~3.92 sec). This indicates that lowering the threshold allows LSH to consider more potential pairs without a significant sacrifice in speed.

However, for a threshold of 0.2, the Fraction drops to 0% despite the fact that the Query Time increases dramatically (~21.56 sec). This paradoxical result can be interpreted as follows: a very low threshold leads to overly "loose" buckets, where pairs with minimal or zero similarity are included. Consequently, when applying the Brute Force Jaccard method to these pairs, no common files are found, which explains the zero Fraction.

New Permutations Test

After the first test with various permutation values and a test with different threshold values, we now perform a new test with more permutations and a fixed threshold value of 0.5, which yielded the highest number of common files. The table below shows the execution times (Build Time, Query Time, and Total Time), the percentage of common similar files between the two methods, and finally the parameters on which the code was executed.

Type	Build Time (s)	Query Time (s)	Total Time (s)	Fraction	Parameters
B.F. - Jaccard	0.00	9403.71	9403.71	100%	K = 7
LSH - Jaccard	3927.45	4.87	3932.32	6%	Perm = 128, Thres = 0.5, K = 7
LSH - Jaccard	4340.56	6.49	4347.05	5%	Perm = 256, Thres = 0.5, K = 7
LSH - Jaccard	5343.29	9.77	5353.05	5%	Perm = 512, Thres = 0.5, K = 7

As shown in the table, increasing the number of permutations from 128 to 512 leads to a significant increase in Build Time (from 3927.45 sec to 5343.29 sec), while the Fraction decreases slightly (6% → 5%), and the Query Time increases slowly but steadily (from 4.87 sec to 9.77 sec).

For permutations = 128, LSH-Jaccard maintains the Fraction at 6%, similar to the previous test (permutations = 64, threshold = 0.5). However, with permutations = 256 and 512, the Fraction drops to 5%, despite the increased computational complexity. This may indicate that increasing the number of permutations does not necessarily improve the accuracy in finding common files when the threshold remains constant. On the contrary, it appears to introduce a slight overfitting in the MinHash signatures, resulting in the exclusion of some truly similar pairs.

At the same time, the Query Time increases due to the greater complexity in managing the LSH buckets, yet it remains extremely low compared to the Brute Force approach (9.77 sec vs. 9403.71 sec). The minimal increase in Query Time (~5 sec for 512 permutations) confirms that LSH still dramatically reduces the number of candidate pairs, even with a higher number of permutations.

Testing with More and Fewer Neighbors

After testing different permutation values and thresholds, we now examine how the algorithms behave when searching for the 5, 7, and 9 nearest neighbors. For the remaining parameters, we kept the combination that provided the best percentage of common neighbors in the shortest time—namely, permutations = 64 and threshold = 0.5. The table below shows the execution times (Build Time, Query Time, and Total Time), the percentage of common similar files between the two methods, and finally the parameters on which the code was executed.

Type	Build Time (s)	Query Time (s)	Total Time (s)	Fraction	Parameters
B.F. - Jaccard	0.00	9394.14	9394.14	100%	K = 5
LSH - Jaccard	3225.46	3.80	3429.26	7%	Perm = 64, Thres = 0.5, K = 5
B.F. - Jaccard	0.00	9362.30	9362.30	100%	K = 7
LSH - Jaccard	3420.37	3.79	3424.15	6%	Perm = 64, Thres = 0.5, K = 5
B.F. - Jaccard	0.00	9362.73	9362.73	100%	K = 9
LSH - Jaccard	3436.73	3.77	3440.50	5%	Perm = 64, Thres = 0.5, K = 9

From the table above, it can be observed that increasing the number of nearest neighbors (K) from 5 to 9 negatively affects the percentage of common similar files (Fraction) between LSH-Jaccard and Brute Force, with the Fraction decreasing from 7% to 5%. At the same time, the execution times (Build Time, Query Time) remain practically unchanged, which indicates that the performance of LSH is not significantly affected by the number of neighbors requested; rather, its accuracy critically depends on its ability to capture high-similarity pairs.

For K = 5, LSH-Jaccard achieves the highest Fraction (7%), since the 5 nearest neighbors correspond to pairs with relatively high Jaccard similarity (≥ 0.5), which LSH can reliably detect. However, when K increases, the Brute Force Jaccard method includes neighbors with lower similarity (< 0.5), which LSH ignores due to the fixed threshold. This explains the gradual decrease of the Fraction to 6% for K = 7 and to 5% for K = 9.

The Query Time remains steady (~3.8 sec) for all values of K, as LSH continues to check the same small subset of candidate pairs from the buckets. In contrast, the Build Time increases slightly (from 3,225.46 sec for K = 5 to 3,436.73 sec for K = 9), possibly due to the creation of more complex structures for storing more neighbor information. However, this increase is minimal, which confirms that the Build Time is primarily determined by the number of permutations.

Compared to the previous experiments, the choice of permutations = 64 and threshold = 0.5 still offers the best balance between speed and accuracy. The instability of the Fraction with larger K is not due to problems with the algorithm, but rather to the nature of the data: the more neighbors requested, the more pairs with low similarity are included by the Brute Force method, which LSH cannot detect due to the strict threshold.

Combination of LSH and Brute Force

Given that the MinHash LSH class from the datasketch library returns all objects that are in the same "bucket", and from these we randomly take a number equal to the number of nearest neighbors, the Brute Force Jaccard method was applied among the objects in the same bucket for selecting the closest objects. For the permutation and threshold parameters, the combination that provided the best percentage of common neighbors in the least amount of time was retained, namely permutation = 64 and threshold = 0.5. As for the number of nearest neighbors, we kept the original value, that is, K = 7.

Type	Build Time (s)	Query Time (s)	Total Time (s)	Fraction	Parameters
B.F. - Jaccard	0.00	9429.46	9429.46	100%	K = 7
LSH - Jaccard	7363.49	194.17	7557.66	7%	Perm = 64, Thres = 0.5, K = 7

From the table above, it can be observed that changing the neighbor selection strategy from random sampling to applying Brute Force Jaccard on all the items in the bucket significantly affects the performance of the LSH-Jaccard algorithm. Despite using the same parameters (permutations = 64, threshold = 0.5, K = 7), the Query Time increases dramatically from 3.8 sec to 194.17 sec, while the Fraction improves only slightly from 6% to 7%. This substantial increase in search time is due to the fact that now all pairs within the bucket are checked using Brute Force, instead of a random subset.

At the same time, the Build Time doubles (from 3420 sec to 7363 sec), possibly due to additional computations or changes in the bucket structure when the algorithm is prepared for a full check of all pairs. However, the minimal improvement in the Fraction (only +1%) indicates that even with an exhaustive check, a threshold of 0.5 remains too strict to capture a significant percentage of truly similar pairs. This suggests that most pairs with similarity ≥ 0.5 are already detected with the previous approach (random sampling), and the new strategy mainly adds false positives or pairs with borderline similarity.

The comparison with Brute Force Jaccard confirms that LSH, even with a full bucket check, remains faster overall (7557 sec vs. 9429 sec). However, this difference is dramatically reduced compared to the previous approach (where LSH was 3 times faster), without a corresponding benefit in accuracy.

Optimal test

The final test involves selecting the optimal parameter values from the above tests so that we can observe what similarity ratio will result. Thus, permutation = 64, threshold = 0.5, and K = 5 were chosen. The combination of the MinHash LSH class with Brute Force Jaccard was used.

Type	Build Time (s)	Query Time (s)	Total Time (s)	Fraction	Parameters
B.F. - Jaccard	0.00	8669.34	8669.34	100%	K = 5
LSH - Jaccard	10167.10	354.43	10521.52	9%	Perm = 64, Thres = 0.5, K = 5

As expected, the best performance is achieved in this case (9%). The implementation time for MinHash LSH (Total Time) remains increased since both the Build Time and Query Time are higher. The implementation time for Brute Force Jaccard is reduced, which is quite logical given that we are searching for fewer neighbors.

Conclusions

The investigation of the performance of the MinHash-LSH algorithm compared to the Brute Force Jaccard highlighted key principles and practical challenges in parameter optimization for a balance between speed and accuracy. The combination of permutations = 64, threshold = 0.5, and K = 7 emerges as the optimal setting for this specific dataset, offering a total execution time three times lower (~3400 sec) than Brute Force (~9400 sec), with a percentage of common similar files (Fraction) at 6%. This means that LSH-Jaccard identifies 6% of the pairs that are considered similar by the precise algorithm, avoiding the exponential time of Brute Force.

The threshold proves to be the most critical parameter. Values such as 0.8 drastically restrict the results to high-similarity pairs (Fraction 2%), whereas a value of 0.5 allows for better coverage (6%) without significant sacrifice in speed. Conversely, very low values (e.g., 0.2) result in invalid outcomes (0% Fraction) due to an excessive influx of unrelated pairs. At the same time, increasing the number of permutations beyond 64 dramatically increases the Build Time (+115% for 128 permutations) without a corresponding improvement in the Fraction, which underscores that the accuracy of the MinHash signatures reaches a saturation point around this value.

The effect of the number of neighbors (K) is equally significant. For K = 5, the Fraction reaches 7%, but it gradually decreases as K increases (5% for K = 9), since the Brute Force method includes neighbors with lower similarity that LSH ignores. This indicates that LSH is optimally tailored for searches that emphasize high-quality results, rather than quantity.

The Brute Force test on all items within the buckets confirmed that this strategy, although slightly increasing the Fraction (+1%), destroys the main advantage of LSH, which is speed. The Query Time skyrocketed (+5,000%), reaching 194 sec, rendering it impractical for larger applications.

Ultimately, MinHash-LSH does not replace Brute Force but constitutes a powerful approximate solution for applications where speed takes precedence over absolute accuracy. The selection of parameters should be based on the specific goals and the nature of the data, with the threshold playing the primary role in balancing recall and precision.

Question 3: Implementation of Dynamic Time Warping (DTW) for Time Series Comparison

Introduction

Dynamic Time Warping (DTW) is a widely used algorithm for measuring similarity between two time series that may vary in length or speed. In this task, we implemented DTW using dynamic programming to compute the optimal alignment between two sequences, ensuring a minimal cumulative distance.

Implementation Steps

1. Data Loading and Preprocessing

The input dataset (dtw_test.csv) consists of time series data stored in two columns: series_a and series_b. These series are stored as string representations of lists. To prepare the data:

- We used the pandas library to load the dataset.
- The ast.literal_eval() function was employed to convert string-based lists into structured numerical arrays using numpy.
- This step ensures that the time series data is in an appropriate format for computational processing.

2. DTW Algorithm Implementation

The DTW distance computation was implemented using a dynamic programming approach, following these key steps:

- Cost Matrix Initialization: A cost matrix (dtw_matrix) of dimensions (n+1, m+1) is initialized with infinite values (np.inf), except for the starting position (0,0), which is set to 0. The extra row and column ensure correct boundary handling by allowing calculations to be performed without requiring special conditions for edge cases. This setup simplifies the implementation and guarantees proper indexing when accessing previous values in the matrix.
- Distance Calculation: The function iterates over the matrix, computing the absolute difference between corresponding points of the two series.

- Cost Update Rule: The cumulative DTW cost at each position (i, j) is determined using the minimum accumulated cost among the three possible transitions:
    1. Insertion (move down in the matrix, dtw_matrix[i-1, j])
    2. Deletion (move right, dtw_matrix[i, j-1])
    3. Match (move diagonally, dtw_matrix[i-1, j-1])
- Final Distance Extraction: The DTW distance is retrieved from the bottom-right cell of the matrix (dtw_matrix[n, m]).

3. Computing DTW Distances

For each row in the dataset:

- The time series pairs (series_a, series_b) were passed to the dtw_distance function.
- Execution time for each DTW computation was recorded to assess performance.
- The computed DTW distances were stored in a list, and progress messages were printed during execution.

4. Performance Evaluation

To evaluate the efficiency of our implementation:

- Total execution time for all DTW computations was measured.
- Average computation time per DTW calculation was computed.
- Given that DTW has a time complexity of , where and are the lengths of the two time series, we observed that execution time increased with longer sequences.

5. Saving Results

- The computed DTW distances were added to the DataFrame.
- The results were saved to a CSV file (dtw_results.csv) in the required format.

Conclusion

The DTW algorithm effectively measures the similarity between time series of different lengths by using dynamic programming for optimal alignment and accurate comparison. The results of the similarity computations between the two series are stored in the file dtw_results.csv and the source code is dtw.py.