Part 1. Text Classification.

In the first part of the project, we aim to explore and compare the performance of two classifiers: Support Vector Machines (SVM) and Random Forests. The process begins with downloading and examining the dataset, followed by preprocessing the text data using the Bag of Words (BoW) representation. To assess the models, we will perform a 5-fold cross-validation to determine which classifier performs best. Lastly, the selected best model will be used to make predictions on the test data. This structured workflow ensures a robust evaluation of the classifiers and their applicability to real-world scenarios.

The Dataset.

The dataset focuses on categorizing news articles into four distinct categories: Business, Entertainment, Health, and Technology. It is organized into two CSV files with the following characteristics:

- train_set.csv (111.795, 4): this file is used for training the classification models. It contains:
  - Id: A unique identifier for each article.
  - Title: The title of the article.
  - Content: The body text of the article.
  - Label: The category to which the article belongs.
- test_set.csv (47.912, 3): this file is used for testing the models on unseen data. It includes the same fields as the training file, except for the Label field, which needs to be predicted.

The dataset is available on Kaggle and serves as the basis for this competition.

Preprocessing the Training Dataset.

The preprocessing of the training dataset involves combining the Title and Content columns into a single text feature and converting it into a numerical format using the Bag of Words (BoW) model. This step is crucial, as machine learning algorithms require numerical inputs. By merging the Title and Content columns, the preprocessing results in a more comprehensive representation for classification by taking into account all the provided related information. Additionally, this combination captures contextual information from both columns, improving the quality of the features for downstream tasks and ensuring that no relevant textual information is overlooked during feature extraction.

The BoW model, widely used for text-based machine learning, often produces sparse matrices where most elements are zero. These matrices are particularly memory-efficient, making them suitable for handling large datasets like ours. This efficiency reduces memory consumption and speeds up computations without losing critical feature information, ensuring a robust and scalable preprocessing pipeline for classification tasks.

As a final step in preprocessing, we convert the categorical labels in the training data into numerical values using a label encoding approach. This transformation is necessary, as ML algorithms also require numerical representations for target variables. The LabelEncoder is used to assign a unique integer to each category, creating a mapping between the original class names and their encoded values. For instance, "Entertainment" might be mapped to 0, "Technology" to 1, and so on. This encoding ensures

that the labels are in a format suitable for training classification models and simplifies the model's interpretation of the target variable during the training process.

Bag of Words (BoW) Representation.

The Bag of Words (BoW) model is used to transform the textual data into numerical feature vectors. This widely-used representation captures the frequency of tokens (words) in a document, ignoring grammar and word order, making it suitable for many text-based machine learning tasks. The process includes the following steps:

- Tokenization: The combined text is split into individual words (tokens).
- Removing common words (stop words): Common English words, such as "the", "is", and "and", are removed from the text. These stop words generally provide little value for classification tasks and are excluded.
- Limiting Vocabulary: When no limitations are imposed on the vocabulary, the total number of unique tokens (words) across the combined Title and Content columns is 241,097. This reflects the full vocabulary size without removing rare or infrequent words. To ensure computational efficiency (and reduce overfitting), the vocabulary size is restricted to the top 10,000 most frequent words. This is achieved by focusing only on the most relevant tokens in the dataset while discarding less common words.
- Word count calculation: The occurrences of each word in the limited vocabulary are counted for every sample, capturing the frequency of meaningful words.
- Vector representation: Each sample is represented as a numerical vector of size 10.000, where each element corresponds to the frequency of a specific word from the vocabulary.

Example of Bag of Words.

Consider the two text samples:

- Sample 1: "Netflix is coming to cable boxes, and Amazon is now your grocery overlord."
- Sample 2: "Pharrell, Iranian President React to Tehran 'Happy' Arrests."

After tokenization, stop word removal, and vocabulary limitation, the resulting vocabulary might look like this (hypothetical example):

["Netflix", "Amazon", "cable", "boxes", "Pharrell", "React", "Tehran", "Happy", "Arrests"]

The BoW representations for the samples would then be:

- Sample 1: [1,1,1,1,0,0,0,0,0]
- Sample 2: [0,0,0,0,1,1,1,1,1]

Count Vectorizer vs. Hashing Vectorizer.

Both Count Vectorizer and Hashing Vectorizer are widely used methods for transforming text into numerical feature vectors. The Count Vectorizer builds a vocabulary of unique tokens (words) from the dataset and maps each token to a feature, representing its frequency in each document. This method

ensures interpretability, as each feature corresponds directly to a token, making it easier to analyze and debug. However, it requires memory to store the vocabulary, which may become expensive for large datasets with high-dimensional feature spaces.

On the other hand, the Hashing Vectorizer uses a hash function to map tokens to feature indices in a fixed-sized vector, without storing the vocabulary. This approach is more memory-efficient and faster, especially for large datasets or streaming applications. However, it introduces potential hash collisions (where two different tokens map to the same feature), leading to a small loss of information, and the features are not interpretable due to the absence of a token-to-feature mapping.

For this project, we chose Count Vectorizer with a limited vocabulary of the top 10.000 most frequent tokens. This decision balances computational efficiency and interpretability, as the restricted vocabulary reduces memory usage while retaining the ability to analyze and understand the relationship between tokens and features. This makes it particularly suitable for our classification task, as it allows us to analyze the contribution of specific words to the model's predictions with a descent and high accuracy.

Resulting Shapes and Vocabulary Size.

- Number of unique tokens in the full vocabulary (without stopwords): 241.097
- Number of unique tokens in the restricted vocabulary (with max_features=10000): 10.000
- Shape of X_train_bow**:** (111.795, 10.000)
- Shape of X_test_bow**:** (47.912, 10.000)

Importance of Preprocessing.

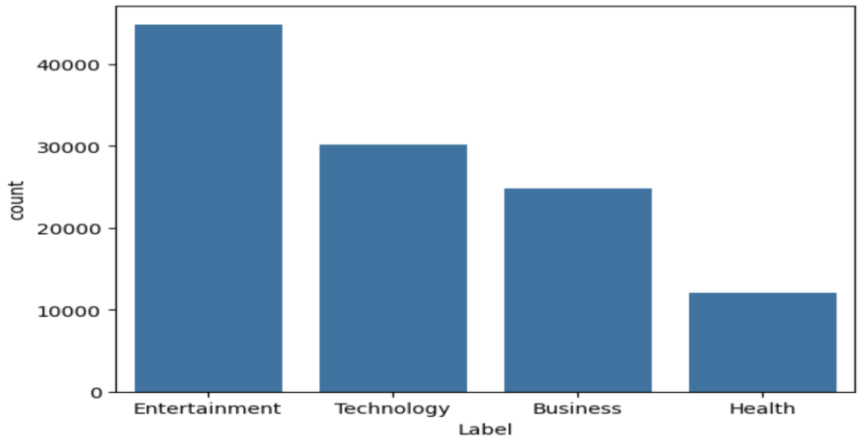
Combining the Title and Content columns ensures that the input data captures both concise and detailed textual information. Using the Bag of Words model, we efficiently convert text into numerical representations that machine learning algorithms can process. By limiting the vocabulary size to the top 10.000 tokens, we strike a balance between computational efficiency and preserving meaningful features for classification, while also reducing noise and overfitting. This preprocessing step is a foundational component of the classification pipeline.

Distribution of Training Data.

Understanding the distribution of the training data is crucial, as it provides insight into potential class imbalances, which can significantly influence model performance and guide the choice of preprocessing and evaluation strategies. Analyzing the data distribution ensures that the model is trained fairly across all classes and helps identify the need for techniques to address imbalances, such as class weighting or sampling methods.

The bar chart below shows the distribution of samples across the four categories in the training dataset. The dataset contains a total of 111.795 samples, distributed as follows:

- Entertainment: 44.834 samples
- Technology: 30.107 samples
- Business: 24.834 samples
- Health: 12.020 samples

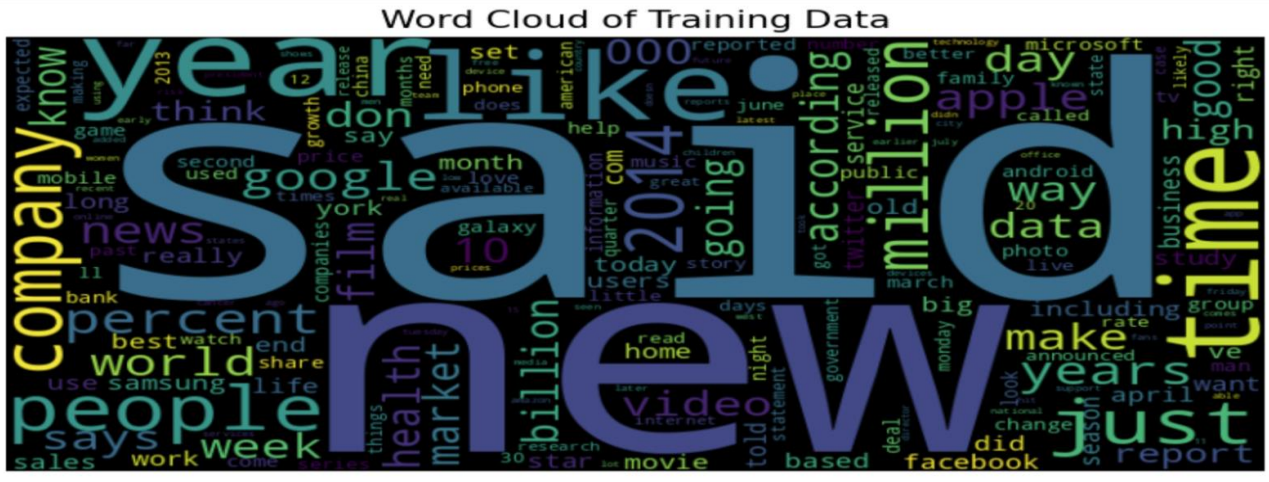


This distribution highlights a noticeable imbalance in the dataset. The Entertainment category has the largest number of samples, accounting for approximately 40% of the total data, while the Health category has the fewest samples, with only about 11%.

The class imbalance may lead to biased model predictions, favoring categories with more samples (e.g., Entertainment) over those with fewer samples (e.g., Health). To address this imbalance, techniques such as class weighting, oversampling, or evaluating the model using metrics like precision, recall, and F1-score will be employed. These measures ensure that the classification performance is assessed fairly across all categories. Including this analysis offers a clear insight into the dataset's structure, enabling a better understanding of its characteristics. It also helps to justify the choices made during preprocessing and the selection of appropriate evaluation metrics and strategies in the model development phase.

Word Cloud of Training Data.

The word cloud below visualizes the most frequently occurring words in the combined Title and Content columns of the training dataset. Larger words, such as "said," "new," "time," "company," and "year," represent higher frequencies within the dataset. These terms often appear in multiple articles, reflecting common language patterns and topics across the dataset.



Some insights:

- The prominence of words like "company," "percent," and "million" suggests a significant focus on business and financial topics, which is likely due to the presence of articles from the "Business" category.
- Words like "health," "film," and "google" highlight content relevant to specific categories, such as "Health" and "Entertainment."
- Common words such as "said" and "new" are not category-specific but appear frequently due to their general usage in reporting and writing.

This visualization helps to better understand the overall composition of the dataset, guiding decisions for feature selection and preprocessing. For example, frequently occurring neutral words like "said" might influence classification and maybe we should not properly account it during text vectorization. This highlights the importance of techniques like stop word removal to focus on words that are more meaningful for distinguishing between categories.

Support Vector Machines (SVM).

Support Vector Machines (SVM) is a powerful supervised learning algorithm commonly used for classification tasks. SVM works by finding the optimal hyperplane that best separates the data into different classes. This hyperplane maximizes the margin between data points from different categories, ensuring robust classification. SVM is effective for both linear and non-linear data, as it can use kernel functions to map non-linear relationships into higher dimensions where the data becomes linearly separable. Additionally, SVM handles high-dimensional data efficiently, making it suitable for text classification tasks with sparse feature spaces. Its versatility and strong theoretical foundation make it a popular choice for various machine learning tasks.

Random Forest.

Random Forest is an ensemble learning method that builds multiple decision trees during training and combines their outputs to improve accuracy and reduce overfitting. Each tree in the forest makes a prediction, and the final output is determined by majority voting for classification tasks. This approach increases model robustness and handles complex, non-linear data effectively. Random Forest is highly interpretable, providing insights such as feature importance, which can help understand the factors influencing predictions. Its ability to handle missing data and work well with imbalanced datasets further enhances its utility in real-world applications.

Ease of Use with Scikit-Learn.

The Scikit-learn library makes implementing both SVM and Random Forest straightforward and efficient. With its intuitive API, users can quickly initialize, train, and evaluate these models using a few lines of code. Scikit-learn also provides built-in tools for preprocessing, cross-validation, and hyperparameter tuning, enabling seamless integration into the machine learning pipeline. Its extensive documentation and active community further simplify the learning curve for new users. This ease of use allows practitioners to focus on experimentation and analysis rather than implementation details, making Scikit-learn an excellent choice for developing machine learning models.

Training SVM and Random Forest without Batches.

Although batch processing can potentially speed up model training, we did not use it in our cross-validation and hyperparameter tuning processes for Support Vector Machines (SVM) and Random Forests due to their algorithmic design and practical limitations.

Standard SVMs, as implemented in libraries like Scikit-learn, rely on solving a quadratic optimization problem to identify the hyperplane that maximizes the margin between classes. This process requires the entire dataset to be loaded into memory and considered simultaneously. The optimization is global and does not natively support mini-batch or incremental updates. While approximate methods or specialized libraries exist to enable batch-like processing, they are not compatible with our goal of conducting precise cross-validation and hyperparameter tuning using the full dataset.

Similarly, Random Forests train an ensemble of decision trees independently on bootstrap samples of the dataset. Each tree uses the entire subset of data for splitting and cannot be updated incrementally without retraining from scratch. While strategies like limiting the number of trees or using approximate algorithms could mimic batch processing, such modifications would deviate from standard Random Forest implementations and potentially impact the consistency of results during cross-validation and tuning.

For these reasons, we chose to use the entire dataset for training in each fold of cross-validation and during hyperparameter tuning to ensure consistency, accuracy, and reproducibility of the results. Although this approach may be computationally intensive, it aligns with the rigorous evaluation required for these models.

The method: Cross-Validation and Hyperparameter Tuning.

To evaluate and optimize our models, we applied **5-fold cross-validation** in combination with **grid search** to identify the best hyperparameter configuration for each model. In 5-fold cross-validation, the dataset is divided into five equal parts (folds). The model is trained on four folds and tested on the remaining fold, repeating the process five times so that each fold serves as the test set once. This approach provides a reliable estimate of the model's performance while minimizing overfitting and ensuring that the evaluation is not biased by a single train-test split.

Grid search was used to systematically explore combinations of hyperparameters, such as the kernel type and regularization strength for Support Vector Machines (SVM) and the number of trees for Random Forest. Ideally, we would have liked to try an extensive range of hyperparameters, including various kernel types (linear, radial basis function, and polynomial), regularization values, and gamma parameters for SVM, as well as different tree-specific parameters for Random Forest, such as maximum depth, minimum samples for splitting and leaf nodes, and feature selection strategies. However, due to limited computational resources and time**,** we restricted our grid search to smaller sets of hyperparameters.

For SVM**,** we tried a linear kernel with regularization values C set to [0.1, 1, 10]. For Random Forest**,** we varied the number of estimators of 50, 100 and 200. While this focused grid search limited the range of experiments, it enabled us to efficiently identify optimal hyperparameters within a reasonable timeframe.

Given the size of the dataset (111.795 samples), we used a subset of 30% of the data for cross-validation and hyperparameter tuning. This subset was randomly selected while preserving the class distribution to ensure representativeness. Using a subset significantly reduces computational costs, allowing us to perform grid search and cross-validation in a practical timeframe without compromising the quality of the results. It ensures that the hyperparameter optimization process is efficient while still providing a robust estimate of the model's performance. Once the best hyperparameters were identified on the subset, we will use them to retrain the final model on the entire dataset in order to maximize its learning potential.

To address class imbalance in the dataset, we applied **class weighting** when initializing both the SVM and Random Forest classifiers. This adjustment ensured that the models gave more importance to underrepresented classes, improving their classification performance across all categories.

Additionally, to speed up the grid search process, we enabled **parallelism** by utilizing multiple CPU cores to use in case of any GPU is not available. This allowed the evaluation of multiple hyperparameter combinations simultaneously, significantly reducing computation time for large datasets or complex models.

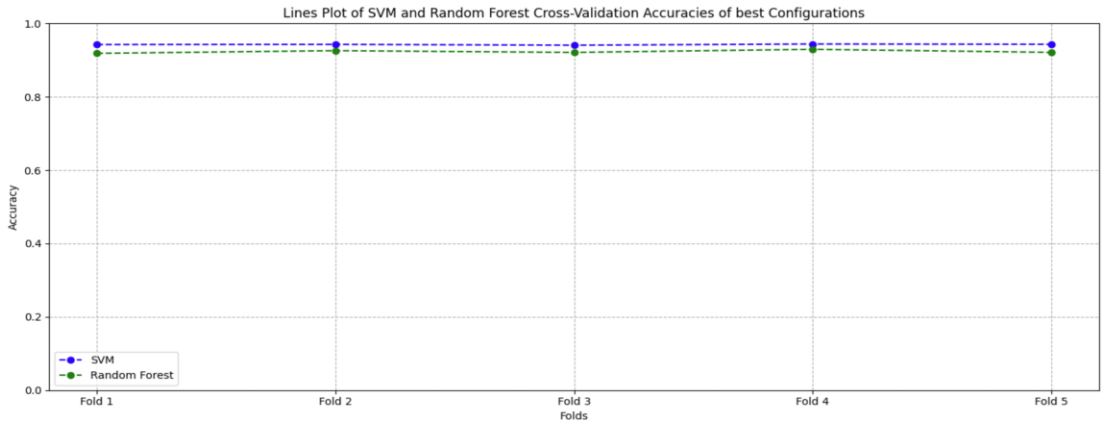
After identifying the optimal hyperparameters and the best-performing model through cross-validation and grid search, we retrain the final model on the entire training dataset. This ensures that the model utilizes all available training data, maximizing its capacity to learn and improving its generalization ability when making predictions on the test data. This two-step process of validation and final training ensures a robust and well-tuned model for the classification task.

Evaluation Approach.

Ideally, we aimed to evaluate our predictions on a separate validation set created from the given training dataset. This would allow us to calculate additional metrics such as the classification report, precision, recall, and F1-score per class, providing deeper insights into the model's performance and its ability to handle imbalances or difficult-to-classify categories. However, since this is a Kaggle competition where the true labels of the test dataset are not provided, we decided to utilize the entire training dataset to train our model. This ensures that the model has access to the maximum amount of data for learning, increasing its potential to generalize better and make more accurate predictions on the unseen test dataset. This approach aligns with the competition's goal of generating the best possible predictions for evaluation on Kaggle's platform.

Results.

In the below figure, we see how the SVM and Random Forest models performed across the 5-fold cross-validation process for their best hyperparameter configurations. The line graph highlights the fold-by-fold accuracies, providing a clear visualization of the overall performance of both models.



The results demonstrate that SVM with the "linear" kernel achieved a marginally higher mean cross-validation accuracy of 94.27% compared to Random Forest, which achieved 92.33%. The consistent performance of SVM across all folds, as shown in the graph, highlights its robustness and reliability. Similarly, Random Forest exhibited stable fold accuracies, with no significant deviations across the folds.

In terms of computational efficiency, Random Forest completed hyperparameter tuning in almost 391 seconds, significantly faster than SVM, which required 614 seconds. This substantial difference in runtime showcases Random Forest's disadvantage in time-sensitive scenarios where computational efficiency is critical.

Overall, while SVM demonstrated slightly superior accuracy, Random Forest provides a competitive and efficient alternative with stable performance across folds, making both models suitable.

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**
Accuracy	94.27 %	92.33 %

Based on the superior accuracy and consistent performance demonstrated by SVM, we selected it as the final model for this task. The predictions on the given test dataset were generated using the best-performing SVM configuration, saved in a CSV file, and subsequently uploaded to the Kaggle competition for evaluation. Below, we also explore a simple neural network model to further analyze its performance on this task.

Neural Network Model.

For task of text classification, and while this goes beyond the scope of the original question, we also implemented a feedforward neural network using the Keras library. The model was designed to handle multi-class classification with an input layer of size equal to the feature set, three hidden layers of decreasing size (256, 128, and 64 neurons respectively), and a final softmax output layer for classification into four classes. Batch normalization was employed after each hidden layer to stabilize and speed up training, while dropout layers (set to 0.3) were added to reduce the risk of overfitting. The model was compiled using the Adam optimizer and categorical cross-entropy loss, making it suitable for multi-class problems.

To enhance training efficiency and improve generalization, we used callbacks**:**

- EarlyStopping was used to stop training when the validation loss stopped improving for five consecutive epochs, ensuring the model did not overfit and saving computational resources.
- ReduceLROnPlateau dynamically adjusted the learning rate by reducing it by a factor of 0.5 if the validation loss improved, enabling finer weight updates for better convergence.

The neural network model was trained for 7 out of the maximum 50 epochs, with early stopping applied to prevent overfitting. A batch size of 64 was used, and the data was split into 80% for training and 20% for validation. This configuration achieved a validation accuracy of 97% with a total training time of 58.24 seconds, demonstrating the effectiveness of the neural network for the given classification task. However, it is noteworthy that the best SVM model, trained on a third of the entire dataset, achieved a validation accuracy of 94.27%, while to be trained on the entire dataset needed much longer training time of 6237.45 seconds. This highlights the trade-off between computational efficiency and accuracy when comparing the neural network to the SVM for this task.

Part 2. Nearest Neighbor Search with Locally Sensitive Hashing.

In the second part of the project, we implement the k-NN algorithm with k=7, beginning with the brute-force approach. This method involves comparing each document in the test set with all documents in the training set using the Jaccard similarity, which measures the ratio of the intersection to the union of their sets. To improve efficiency, we then apply the Locality Sensitive Hashing (LSH) technique. The LSH approach first identifies candidate pairs of test and training documents that are likely to exceed a predefined Jaccard similarity threshold. The actual similarity is computed only for these candidate pairs, drastically reducing the computational cost while maintaining accuracy.

The Dataset and Preprocessing.

We will use the same dataset as in the previous part, with minimal additional preprocessing. Each sample in the dataset is represented by tokens (words) and their corresponding frequencies. To calculate the similarity between samples using the Jaccard similarity function, we transform the train and test datasets into a set-based format, where each sample is converted into a set of token indices corresponding to the tokens present in the sample. This transformation enables the Jaccard similarity function to compute the overlap between sets effectively, as it is specifically designed for set representations. The Jaccard similarity is defined as the size of the intersection divided by the size of the union of two sets, yielding a score between 0 (indicating no overlap) and 1 (indicating complete overlap). These transformed datasets are then utilized with the custom Jaccard similarity function to evaluate the similarity between any pair of samples, ensuring a straightforward and computationally efficient approach.

K-NN (Brute-Force method).

We applied the brute-force method to identify the top K-nearest neighbors, where K=7 for each test sample using the Jaccard similarity measure. In this process, we iterated over all test and training samples, calculating the similarity between each test sample and every training sample. The Jaccard similarity was used to measure the overlap between sets of token indices for the test and training samples.

For each test sample, the similarities with all training samples were computed and ranked in descending order. The indices of the K-most similar training samples were extracted and stored as the nearest neighbors for the corresponding test sample. The resulting indices indicate the positions of the most similar training samples in the training dataset.

For instance, if the indices for a test sample are [99,36,26,27,28,29,30], this means the training sample at index 99 has the highest similarity to the test sample, the one at index 36 has the second-highest similarity, and so on, with the sample at index 30 being the seventh most similar.

This brute-force approach ensures 100% accuracy in identifying the exact K-nearest neighbors because it exhaustively computes the Jaccard similarity for every pair of test and training samples. However, this exhaustive computation comes at the cost of time and computational resources (see it at the results section). As a result, the brute-force method serves as a benchmark for comparison with more efficient but approximate methods, such as Locality-Sensitive Hashing (LSH). These approximate methods prioritize speed and scalability by sacrificing some degree of accuracy for significantly faster query times.

Locality Sensitive Hashing (LSH).

Locality-Sensitive Hashing (LSH) is an efficient algorithm designed to approximate similarity searches in high-dimensional datasets. Unlike brute-force methods, which compute pairwise similarities exhaustively, LSH achieves computational efficiency by grouping similar items into the same "buckets" based on their hashed representations. This is accomplished using MinHash signatures, compact fixed-length vectors that approximate Jaccard similarity between sets. LSH works by dividing these signatures into smaller bands and hashing items into buckets within each band. Items in the same bucket are treated as candidates for similarity, significantly reducing the number of comparisons required. LSH is crucial when working with large-scale datasets, as it strikes a balance between accuracy and computational efficiency, enabling faster query times without fully sacrificing precision. Its workflow involves generating MinHash signatures, building an LSH index, and querying it to retrieve candidate neighbors for similarity computations.

K-NN with LSH.

We implemented the LSH method to enhance the efficiency of K-Nearest Neighbor (K-NN) similarity search, using it as an alternative to the computationally expensive brute-force approach. The steps involved were as follows:

- Generating MinHash Signatures**:** For each training sample, we computed MinHash signatures to create compact representations of the data.
- Building the LSH Index**:** These MinHash signatures were inserted into an LSH index, grouping similar items into the same buckets based on a similarity threshold.
- Querying the LSH Index**:** For each test sample, its MinHash signature was computed and used to query the LSH index, retrieving candidate neighbors. These candidates were refined by calculating exact Jaccard similarity with the test sample to determine the top K-most similar documents.
- Comparison with Brute-Force Results**:** Finally, the accuracy of the LSH method was evaluated by comparing its results with the brute-force method using a fraction metric, which calculates the proportion of true K-nearest neighbors identified by LSH relative to brute-force results.

We experimented with various configurations to optimize the LSH method:

- Permutations**:** 16, 32, 64
- Thresholds**:** 0.8, 0.6, 0.5, 0.3

Running the Training Process.

Initially, we began the training process for the k-NN algorithm using the brute-force method in the simplest approach possible: without any parallelism, sparse matrix optimizations, or batch processing. The goal was to establish a baseline implementation and understand the computational demands of the task. During this first attempt, we processed all test samples individually against the entire training dataset, calculating all pairwise similarities in a straightforward nested loop structure. All experiments were conducted in the Google Colab environment.

However, even on a high-performance system equipped with a powerful GPU and 83.5 GB of RAM, this naive approach proved to be both computationally and memory-intensive, requiring nearly 7 hours to complete the training process. This highlighted the need for a more optimized approach. As a result, we shifted to training in batches, experimenting with a batch size of 500 and 32, in order to improve performance and limit resource utilization. But, even with batch processing, the process required significant computational time, taking approximately 6.88 hours to complete, and we observed no substantial improvement in performance for either batch size. All experiments and their results are provided in the attached notebook for further reference and analysis.

For the LSH method, starting with a high threshold (e.g., 0.9) led to errors due to insufficient band creation ($b < 2$). This limitation arises from the relationship between the number of permutations and the similarity threshold. High thresholds restrict the partitioning of the MinHash signature matrix, resulting in insufficient buckets to group similar items effectively. By lowering the threshold to values like 0.8, 0.7, 0.5, or 0.3, we allowed the LSH index to create enough bands to capture similar items in the same bucket, improving the method's ability to identify candidate neighbors. This adjustment avoided computational errors and enabled LSH to balance accuracy and efficiency, achieving faster query times compared to brute force while maintaining robust performance tailored to the dataset's characteristics.

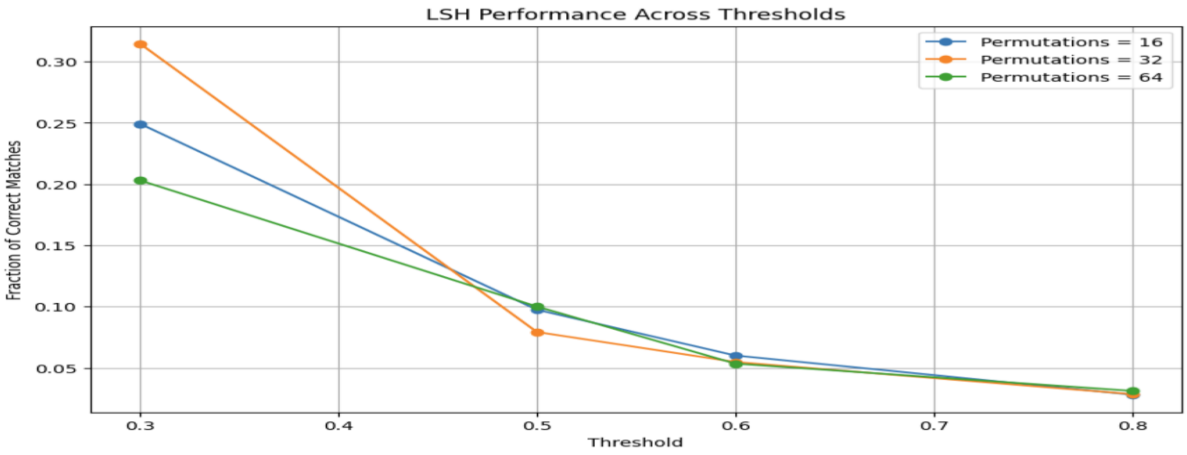
We also experimented with the use of class weights in both brute-force and LSH methods to address the class imbalance in the dataset. The weights were calculated as the inverse of the number of samples in each category, and each sample's similarity score was multiplied by the corresponding weight of its category during the calculations. However, neither approach demonstrated improved performance with the application of these weights. For the brute-force method, class weights had no effect on accuracy, as the approach inherently identifies the true nearest neighbors without bias. In the LSH method, applying class weights resulted in slightly lower fractions of correct matches across all thresholds and permutations compared to results without weights. This suggests that the weight adjustment disrupted the similarity structure or unduly amplified the influence of smaller categories, potentially distorting the overall performance. Despite similar computational costs, these results highlight the limitations of this weight adjustment strategy in both methods.

The build, query, and total times for all combinations of thresholds and permutations are reported below. As expected, the LSH method demonstrates significantly reduced computation times compared to the brute-force approach, reinforcing its efficiency in handling large datasets. However, achieving

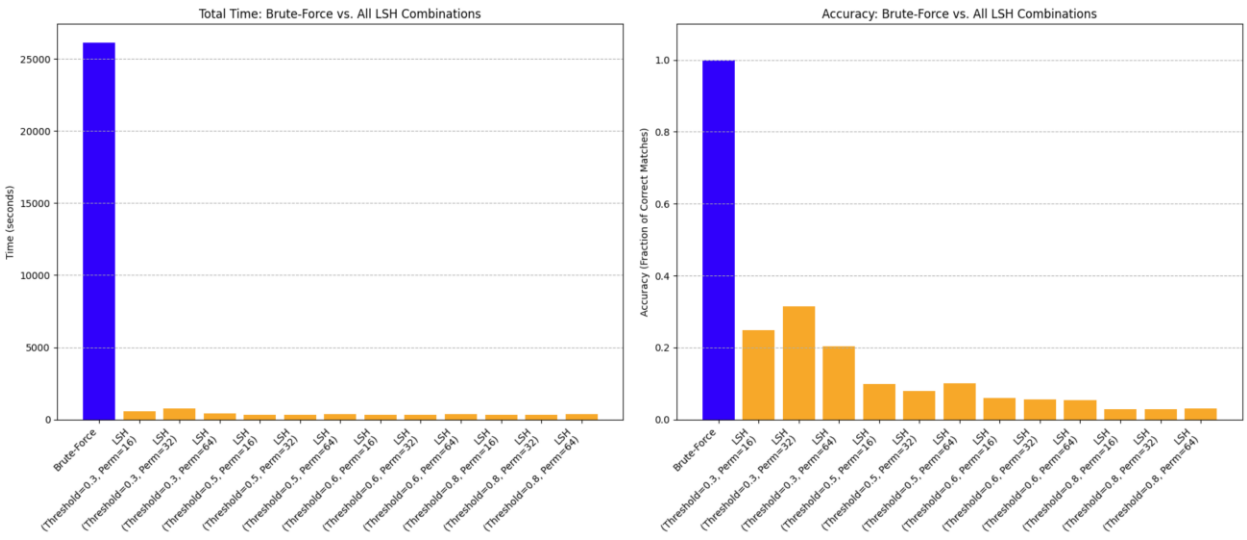
higher accuracy with LSH remains challenging, requiring careful parameter tuning and potential alternative strategies to improve performance.

Results.

To better analyze the performance and trade-offs between brute-force and LSH methods, we provide visual representations that highlight key findings from our experiments.



This plot shows the LSH performance across thresholds for different permutations (16, 32, and 64). The x-axis represents the threshold values, and the y-axis represents the fraction of correct matches. The lines illustrate the trade-off between accuracy and thresholds, with lower thresholds yielding higher fractions of correct matches. Higher permutations generally improve the performance slightly but show diminishing returns at higher thresholds.



This visualization compares the computation times and accuracies between the brute-force method and all LSH combinations of thresholds and permutations. The left subplot highlights the dramatic difference in total computation times, with brute force being significantly slower than all LSH configurations. The right subplot shows the accuracy (fraction of correct matches) for each method, indicating that while

brute force achieves perfect accuracy, LSH performance varies depending on the threshold and permutations, with the highest accuracy observed for lower thresholds and higher permutations.

In the table below, we present a detailed comparison of the brute-force method and the Locality-Sensitive Hashing (LSH) method for K-Nearest Neighbor (K-NN) search, highlighting the performance metrics across different configurations. The table includes columns for Build Time**,** Query Time**,** Total Time**,** and the Fraction of Matching Results between brute-force and LSH methods. Additionally, the table specifies the number of permutations and similarity thresholds used in each LSH configuration.

Type	**Build Time**	**Query Time**	**Total Time**	**BF vs. LSH (Fraction)**	**Permutation**	**Threshold**
Brute-Force Jaccard	0	23882.8	23882.8	100%	-	-
LSH-Jaccard	189.018	87.6272	276.646	2.84%	16	0.8
LSH-Jaccard	201.790	92.2695	294.060	2.87%	32	0.8
LSH-Jaccard	230.711	101.942	332.653	3.15%	64	0.8
LSH-Jaccard	186.935	86.9842	273.919	6.27%	16	0.6
LSH-Jaccard	201.719	92.3166	294.036	5.60%	32	0.6
LSH-Jaccard	227.700	103.902	331.603	5.42%	64	0.6
LSH-Jaccard	185.307	94.6027	279.910	11.08%	16	0.5
LSH-Jaccard	202.449	93.2308	295.680	8.48%	32	0.5
LSH-Jaccard	229.616	105.553	335.169	10.9%	64	0.5
LSH-Jaccard	183.318	344.746	528.065	32.62%	16	0.3
LSH-Jaccard	201.627	557.900	759.528	42.46%	32	0.3
LSH-Jaccard	232.738	162.761	395.499	25.48%	64	0.3

The brute-force method serves as the baseline for comparison, offering a significant total time due to pairwise comparisons across all data points. In contrast, the Locality-Sensitive Hashing (LSH) method provides a more efficient alternative, with different configurations tested using permutations of 16, 32, and 64 alongside thresholds of 0.8, 0.6, 0.5, and 0.3. These parameters introduce trade-offs between computational effort and accuracy. Lower thresholds lead to more buckets, increasing the likelihood of capturing similar items but also introducing potential noise, which can slightly reduce accuracy. Conversely, higher permutations improve the quality of similarity matching by generating longer signature vectors, reducing the probability of collisions (where dissimilar points are treated as similar), but at the cost of increased build and query times.

The table illustrates these trade-offs clearly. The brute-force method, while computationally intensive with a total query time of 23,882.8 seconds, guarantees a 100% match fraction. LSH dramatically reduces the total computation time, achieving substantial time savings (e.g., total times ranging from 276.646 to 759.528 seconds) depending on the parameter configuration. However, the fraction of correct matches varies across these configurations. For example, at a threshold of 0.8, LSH achieves a smaller fraction of correct matches (2.84–3.15%) compared to the brute-force method, while at a lower threshold of 0.3 and with higher permutations, the fraction of correct matches increases significantly, reaching up to 42.46%. This improvement highlights the trade-off: increasing permutations and adjusting the threshold enhances the alignment of LSH with the brute-force results, but it also increases the computational time, as seen in longer build and query times.

In conclusion, the brute-force method provides a benchmark for accuracy, but its computational cost makes it impractical for large datasets. LSH, with its parameterized configurations, strikes a balance between efficiency and accuracy. The choice of parameters, such as permutations and thresholds, depends on the specific requirements of the application, such as the desired accuracy versus available computational resources. The analysis highlights the importance of parameter tuning to optimize the performance of LSH for similarity matching tasks.

Resource Utilization during Training and Evaluation.

Efficient utilization of computational resources played a critical role in training and evaluating machine learning models for Part 1 (Text Classification with SVM and Random Forest) and Part 2 (KNN: Brute Force vs. KNN with LSH). Below is a detailed overview of the resources leveraged during these processes.

- *High-RAM Runtime:* The training and evaluation were conducted in a high-RAM runtime environment, providing 89.6 GB of available RAM**.** This memory was essential for handling computationally intensive tasks such as vectorizing large text datasets, training machine learning models, and performing nearest neighbor searches for KNN. The system RAM utilization peaked at 60.6 GB (brute-force method) out of 83.5 GB, demonstrating the significant memory demands of these operations.
- *GPU Utilization:* The runtime environment included an NVIDIA A100-SXM4-40GB GPU with 40 GB of GPU memory**,** which provided substantial computational power for model training and evaluation. Tasks such as SVM and Random Forest hyperparameter tuning, training, and KNN search with brute force were effectively accelerated by the GPU. GPU memory utilization reached 32.5 GB out of 40 GB during training, highlighting the intensive nature of these operations.
- *Disk Usage:* The disk space utilized remained minimal, with 33.1 GB out of 235.7 GB used during the entire process. This reflects efficient management of intermediate files, model outputs, and datasets.

These computational resources enabled the smooth execution of all experiments in both parts of the project, ensuring the efficient handling of large datasets, complex computations, and time-intensive tasks such as hyperparameter optimization and large-scale nearest neighbor searches.

Applications of LSH.

Having done all the above experiments, we are curious to see where all this is implemented in practice. The Locality-Sensitive Hashing (LSH) method finds applications in various fields requiring efficient similarity search and clustering for high-dimensional data. In the domain of natural language processing, LSH is used to identify duplicate documents, plagiarism detection, and recommendation systems by quickly finding similar text-based content. It is widely employed in image retrieval systems, where similar images are retrieved from large datasets based on perceptual similarity. LSH is also crucial in genomics, where it helps in detecting similar DNA or protein sequences across massive biological databases. Additionally, it plays a significant role in anomaly detection for cybersecurity, efficiently finding patterns that match known malicious activity. Its ability to handle large datasets with reduced computational costs makes it an invaluable tool in real-world applications that demand scalable and approximate nearest-neighbor searches. These examples underscore the relevance and versatility of the LSH method in tackling practical challenges across industries.