Text Classification using Machine Learning

Corentin Latimier
Department of Aerospace Engineering
Polytechnique Montréal
Montreal, Canada
`corentin.latimier@polymtl.ca`

Poobesh Kumar Subramaniam
Department of Mechanical Engineering
McGill University
Montreal, Canada
`poobesh.subramaniam@mail.mcgill.ca`

Abstract

This study applies machine learning to classify Reddit posts from four city-specific subreddits: Toronto, Brussels, London, and Montreal. The task is a multiclass text classification problem with balanced datasets. Advanced preprocessing techniques such as stopword removal, lemmatization, and dimensionality reduction are used to improve feature extraction. We develop a custom scoring function for feature selection, tailored for text analysis. Various models, including Naïve Bayes, Support Vector Machines, and ensemble methods, are evaluated, with model stacking shown to enhance accuracy.

1 Introduction

This project focuses on classifying Reddit posts from four subreddits: Toronto, Brussels, London, and Montreal, each with unique cultural and linguistic features. The dataset includes labeled training and unlabeled test data for a Kaggle competition. Challenges include handling bilingual posts from Montreal and ensuring model generalization. Preprocessing steps like text normalization, stopword removal, and lemmatization are applied, followed by TF-IDF and dimensionality reduction. Several classifiers, including Naïve Bayes, Support Vector Machines, Logistic Regression, and ensemble methods, are evaluated.

2 Presentation of the datasets

2.1 Description of the datasets

This project uses a dataset of Reddit posts from four city-based subreddits: **Toronto**, **Brussels**, **London**, and **Montreal**. The dataset includes a labeled **training set** for model training and an unlabeled **test set** for evaluation, with the test set also used in a Kaggle competition. The goal is to predict the subreddit of origin for a given post or comment, making this a **multiclass classification problem with 4 classes**.

2.2 Analysis of the datasets

The training dataset consists of 1,400 examples, with an equal number of samples per class, ensuring a balanced distribution. The test dataset contains 600 examples. Figure 1 illustrates the distribution of text lengths for both the training and test datasets. We observe that the overall distributions are quite similar, which highlights the fact that the two datasets share comparable characteristics. To gain insights, PCA analysis is performed on the training dataset after applying TF-IDF vectorization (**??**). The vectorizer incorporates both English and French stopwords, as Montreal posts contain both languages, and lemmatization (3) is used to reduce words to their base form. The results are presented in Figure 2 and are visualized in the 2D space formed by the first two PCA components. We observe that a portion of the Montreal class, consisting of French-language entries, distinctly separates from the rest of the dataset.

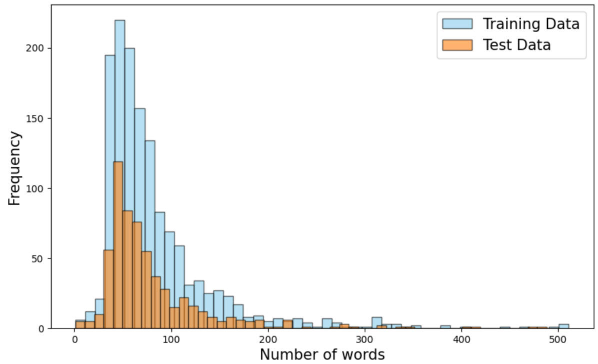


Figure 1: Distribution of text lengths within training and test datasets

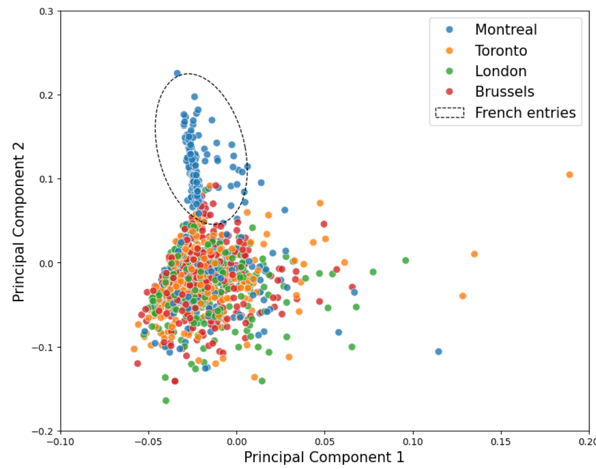


Figure 2: PCA analysis visualization of the training dataset

2.2.1 Dataset pre-processing for features vectors preparation

Both the training and test datasets undergo several preprocessing steps before the posts or comments are transformed into feature vectors. The preprocessing pipeline consists of the following steps:

1. **Text normalization:** The first step is to remove punctuation, capitalization, numbers, and extra spaces in order to obtain lists of lowercase words.
2. **Stopword Removal:** The next step involves eliminating stopwords—common words such as "the," "of," and "about"—which are unlikely to provide valuable information about the content of a document. The stopwords used are sourced from both English and French stopword lists from the NLTK (*Natural Language Toolkit* [6]) module.
3. **Lemmatization [3]:** The third step applies lemmatization, which reduces words to their base form (e.g., "running" becomes "run"). This process helps treat different word forms as a single feature, improving the model's generalization.

This processing pipeline results in a total of 10,207 features. Once the texts have been processed, a vectorization operation is performed to transform the list of words into numerical vectors. Depending on the model used, different vectorization methods can be applied.

Method 1: TF-IDF [8]
TF-IDF (Term Frequency-Inverse Document Frequency) transforms the text data by giving each term a weight that reflects its importance across documents. Specifically, it is given by the formula:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

where $\text{TF}(t, d)$ is the frequency of the term t in the document d, and $\text{IDF}(t)$ is the inverse document frequency, defined as:

$$\text{IDF}(t) = \log\left(\frac{N}{1 + \text{DF}(t)}\right)$$

where N is the total number of documents, and $\text{DF}(t)$ is the number of documents containing the term t.

Method 2: Binary vectors :
For some models, such as Bernoulli Naive Bayes [**BNB**], the feature vectors must be binary, consisting of 0s and 1s. In this method, a value of 1 is assigned when the word is present in the text, and 0 is assigned when the word is absent.

Method 3: Sentence Embeddings :
Sentence embeddings are vector representations that capture the semantic meaning of entire sentences, providing an alternative to word-level representations. They are particularly useful for multilingual data, such as the bilingual content in the dataset (e.g., English and French text). The `SentenceTransformer` model (`paraphrase-multilingual-MiniLM-L12-v2` [7]) generates a 384-dimensional vector for each sentence, representing its meaning. Words or sentences with similar semantic meanings will have smaller Euclidean distances between their respective embeddings. These embeddings are combined with normalized TF-IDF vectors, enhancing the feature set by integrating both statistical and semantic information for improved classification performance.

Depending on the model, if the vectorized texts are non-binary vectors, normalization techniques are applied. Z-score normalization standardizes each feature by removing the mean and scaling to unit variance. Min-Max normalization scales features to a fixed range, typically [0, 1]. L2 normalization scales the vectors so that their Euclidean norm equals 1.

3 Proposed approach

3.1 Dimensionality reduction

To mitigate over-fitting and improve our model's generalization to new data, we applied dimensionality reduction techniques, exploring both feature selection and feature construction methods.

For feature selection, we used variable ranking methods with different scoring functions. In particular, we developed a custom scoring function called **distinctiveness**, defined as:

$$D_j(w) = f_j(w) - \frac{\sum_{k \neq j} f_k(w)}{N-1}$$

where $D_j(w)$ represents the distinctiveness score for word w in class j, $f_j(w)$ is the frequency of w in class j, and N is the total number of classes. The distinctiveness score is higher for features that appear more often in one class than in others. A hyperparameter (k_d) selects the top k_d features per class. To prevent the Montreal dataset from favoring French words, we split it into English and French subsets, ensuring unbiased feature selection for both languages. Figure 4 shows the top 15 distinctive features for each class, highlighting the effectiveness of this scoring function.

To assess the efficiency of this custom scoring function for this problem, we compared model performance based on feature ranking methods using alternative, more traditional feature scoring functions. In particular, we considered the **mutual information** [4] scoring function as a baseline method.

For feature construction method, **Truncated Singular Value Decomposition** [2] (Truncated SVD) was used. Truncated SVD is a dimensionality reduction technique that simplifies a dataset by approximating it with a lower-dimensional representation. It works by decomposing the original matrix into three components: singular values and their corresponding vectors. The technique then keeps only the most significant singular values and their associated vectors, discarding less important ones. This reduces the number of features, maintaining the most relevant information and minimizing computational cost.

3.2 Classifier selection

A brief overview of the motivation behind each classifier as well as their respective evaluation strategies are provided below.

3.2.1 Bernoulli Naive Bayes [BNB]

The only classifier fully implemented from scratch, is Bernoulli Naive Bayes. This classifier is a *generative model* that operates based on Bayes' theorem. It assumes the conditional independence of features given the class label. The model works by learning the prior class probabilities $P(c_j)$ and the conditional probabilities $P(x_i|c_j)$. Class labels are then assigned according to the following rule:

$$Class = \text{argmax}_c \log \left[P(c)\Pi_{j=1}^{m} P(x_j|c) \right]$$

In this study, only the multiclass version of Bernoulli Naïve Bayes was implemented, as the 1-vs-all approach would result in highly unbalanced datasets. Moreover, Laplace smoothing is used to handle cases where certain events have zero probability. By adding a small constant (in this study 1) to each count, it ensures that no probability is ever exactly zero, which stabilizes the model and improves generalization, especially with sparse data.

3.2.2 Multinomial Naive Bayes [MNB]

Multinomial Naive Bayes (MNB) is a generative model based on Bayes' theorem, similar to Bernoulli Naive Bayes (BNB). While both assume feature independence given the class label, MNB is designed for discrete count data like word frequencies, whereas BNB is for binary data indicating feature presence or absence. MNB uses a multinomial distribution for features, making it suitable for tasks like document classification, while BNB is better for binary classification. Laplace smoothing is applied in both models to handle zero probabilities.

3.2.3 Logistic Regression [LR]

Logistic Regression is a *discriminative model* used for binary and multiclass classification. It estimates the probability of a data point belonging to a class using the logistic (sigmoid) function. For binary classification, the probability of the positive class is given by:

$$P(y=1|\mathbf{x}) = \frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$$

where $\mathbf{x}$ is the feature vector, $\mathbf{w}$ represents the model weights, and b is the bias term. The model parameters are learned by minimizing the negative log-likelihood:

$$\min_{\mathbf{w},b} -\frac{1}{N}\sum_{i=1}^{N}[y_i \log P(y_i|\mathbf{x}_i) + (1-y_i)\log(1-P(y_i|\mathbf{x}_i))]$$

To prevent overfitting, L2 regularization is commonly applied by adding a term proportional to $\|\mathbf{w}\|^2$ to the loss function.

3.2.4 Support Vector Machines [SVM]

The objective of Support Vector Machines (SVMs) is to identify a hyperplane that maximizes the margin separating different classes, while permitting a degree of slack to accommodate non-separable data points. This optimization problem, incorporating a specified kernel K to map input features into a higher-dimensional space, is formulated as follows:

$$\begin{aligned} &\min_{\vec{w},b,\vec{\xi}} \|\vec{w}\|_2^2 + \gamma\sum_{i=1}^{N}\xi_i \\ &\text{s.t } y_i\left(\vec{w}^T\Phi(\vec{x}_i)+b\right) \geq 1-\xi_i \;\forall i \\ &\xi_i \geq 0 \;\forall i \end{aligned}$$

Where: $\mathbf{w}$ and b define the hyperplane, ξ_i are the slack variables allowing for misclassifications, γ is the regularization parameter that controls the trade-off between maximizing the margin and minimizing classification errors, $\phi(\mathbf{x})$ is the feature mapping function induced by the kernel, where $K(\mathbf{x}_i,\mathbf{x}_j) = \phi(\mathbf{x}_i)\cdot\phi(\mathbf{x}_j)$. SVMs are highly relevant to our problem as they perform well with high-dimensional data, a typical characteristic of text classification tasks. Moreover, their ability to utilize various kernels enables them to capture complex relationships within the data, even when the classes are not linearly separable, which is undoubtedly the case in the classification problem considered.

3.2.5 Stacking models

Stacking models is an *ensemble learning technique* where multiple base models are trained on the same data, and their predictions are combined by a higher-level model, called the meta-model. The base models make individual predictions, and the meta-model learns how to optimally combine these predictions to improve the overall performance. Stacking works best when the base models have complimentary strengths and weaknesses.

3.2.6 CatBoost [CB] [1]

CatBoost is a boosting model that combines multiple base models to improve performance. It trains models sequentially, each correcting errors from the previous one, focusing on misclassified samples. The final prediction is a weighted sum of all models' predictions. CatBoost is in particular a gradient boosting algorithm designed for classification, minimizing the log loss function:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i\log(p_i) + (1-y_i)\log(1-p_i)]$$

where y_i is the true class label, p_i is the predicted probability, and N is the number of samples. In each iteration, a new decision tree is added to update the prediction:

$$\hat{y}_m = \hat{y}_{m-1} + \eta \cdot \Delta\hat{y}_m$$

where η is the learning rate, and $\Delta\hat{y}_m$ represents the prediction update. CatBoost uses ordered boosting to effectively handle categorical features, updating predictions iteratively.

3.3 Overview of the different approaches

Table 1 provides an overview of the different models considered in this paper.

Table 1: Model configurations summary

Model #	Classifier	Vectorization	$D.R^*$	Normalization	Hyper param.
1	BNB	Binary	Distinctiveness	N/A*	k_d^*, α^*
2	BNB	Binary	Mutual info.	N/A	$k_{M.I}^*$
3	SVM	TF-IDF	Mutual info.	z-score	$k_{M.I}$, γ, K^*
4	Stacking [3]	Bin.,TF-IDF	Mutual info.	N/A - z-score	**2 + 3**
5	MNB	TF-IDF	Truncated SVD	Min-Max	α
6	LR	TF-IDF + Sentence Emb	Truncated SVD	L2	C*, Solver
7	CB	TF-IDF + Sentence Emb	Truncated SVD	L2	itr*, lr*,d*

$\mathbf{D.R}^*$: Dimensionality reduction, $\mathbf{N/A}$: Not applicable, $\mathbf{k_d}$: Number of most distinctive features selected per class, α : Laplace smoothing parameter, $\mathbf{k_{M.I}}$: Number of top features selected based on mutual information, **K**: Kernel type, **C**: L2 regularization parameter, **itr**: early stopping threshold for convergence, **lr**: Learning Rate, **d**: maximum depth of the decision trees.

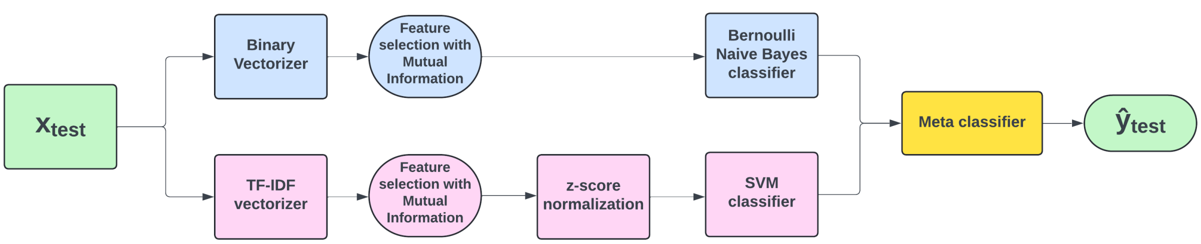


Figure 3: Model #4 definition (stacking model)

3.4 Hyperparameters selection

Hyperparameter selection is carried out using 10-fold cross-validation, with optimal values chosen to maximize validation accuracy and ensure low bias and variance. We assess potential high bias by comparing model accuracy to an estimated human performance ceiling of 90-95% accuracy. Additionally, we aim to identify stable ranges for each hyperparameter, where small adjustments do not significantly affect model accuracy, enhancing robustness and consistency in performance. Table 2, provided in the appendix, displays the optimal hyperparameters determined for each model used in the subsequent analysis.

4 Results and discussion

The model's performance metrics are presented in Table 3 (provided in Appendix), including training accuracy and validation accuracy across 5 or 10-fold cross-validation. For validation accuracy, detailed results are shown for each label to provide insights into which subreddits are more challenging to classify accurately. The table also includes the computing time, representing the total time required to perform 5 or 10-fold cross-validation, encompassing both training and prediction phases.

Models 1 and 2, which differ only in the feature selection scoring function, produce comparable results. While mutual information provides a slight improvement over the distinctiveness-based scoring (+3.8%), it requires more features and parameters. Thus, the distinctiveness scoring function developed is competitive with the current state-of-the-art scoring functions in text analysis.

Both Bernoulli Naïve Bayes models perform well for most classes, except Montreal, where the accuracy is notably lower. Model 3 (SVM) has lower overall accuracy than Bernoulli Naïve Bayes, but excels in predicting Montreal labels, as expected due to the distinct nature of the Montreal dataset, as shown in Figure 2. Therefore, SVM is well-suited for distinguishing Montreal from other classes. Additionally, SVM requires significantly more computational time than the Bernoulli Naïve Bayes model, as it involves solving an optimization problem iteratively, unlike BNB.

Since Models 2 and 3 complement each other, we define Model 4 as a stacking model that combines both. A meta-classifier uses the predictions from both models to generate the final output, as shown in Figure 3. Several experiments were conducted for the meta-classifier (Logistic Regression model, etc.), and the simplest approach proved to be the most accurate:

$$\hat{y} = \begin{cases} \hat{y}_{\text{SVM}} & \text{if} \quad \hat{y}_{\text{SVM}} = \text{Montreal} \\ \hat{y}_{\text{BNB}} & \text{otherwise} \end{cases}$$

We observe that combining both models results in a higher validation accuracy (+2.7%) compared to using Model 2 alone. However, this comes at the cost of significantly increased computation time, which is three times higher than using Model 3 alone, as both models need to be optimized and processed for each prediction. Model 4 is the best performing model we obtained. The confusion matrix for this model 5, provided in the appendix, is nearly diagonal, which indicates strong predictive performance.

Logistic regression (LR) emerged as the most effective classifier among models 5, 6 and 7 with a due to high validation accuracy across all classes and a low variance between training accuracy and validation accuracy. It is also computationally efficient, completing 5-fold cross-validation in 1.24 seconds. CatBoost demonstrated consistent accuracy across all classes (e.g., Toronto: 72.19%, Montreal: 67.36%) but took significantly more time to train (389 seconds). Multinomial Naive Bayes (MNB) was the fastest (3.2 seconds) but had the lowest accuracy overall, particularly struggling with the London and Toronto classes. Although Logistic Regression achieved higher accuracy, CatBoost, despite being more computationally intensive, may still be preferable in certain situations. This is because CatBoost tends to be less sensitive to data preprocessing and normalization than Logistic Regression, making it easier to implement and more effective at generalizing.

5 Statement of contributions

Corentin Latimier worked on models 1, 2, 3, and 4, as well as the distinctiveness scoring function, while Poobesh Kumar Subramaniam concentrated on models 5, 6, and 7.

References

[1] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. *CatBoost: gradient boosting with categorical features support*. 2018. arXiv: 1810.11363 [cs.LG]. URL: https://arxiv.org/abs/1810.11363.

[2] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*. 2010. arXiv: 0909.4061 [math.NA]. URL: https://arxiv.org/abs/0909.4061.

[3] Divya Khyani et al. "An Interpretation of Lemmatization and Stemming in Natural Language Processing". In: *Shanghai Ligong Daxue Xuebao/Journal of University of Shanghai for Science and Technology* 22 (Jan. 2021), pp. 350–357.

[4] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. "Estimating mutual information". In: *Phys. Rev. E* 69 (6 June 2004), p. 066138. DOI: 10.1103/PhysRevE.69.066138. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.066138.

[5] Dong C. Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Math. Program.* 45.1–3 (Aug. 1989), pp. 503–528. ISSN: 0025-5610.

[6] Edward Loper and Steven Bird. *NLTK: The Natural Language Toolkit*. 2002. DOI: 10.48550/ARXIV.CS/0205028. URL: https://arxiv.org/abs/cs/0205028.

[7] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: http://arxiv.org/abs/1908.10084.

[8] "TF–IDF". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 986–987. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_832. URL: https://doi.org/10.1007/978-0-387-30164-8_832.

Appendix

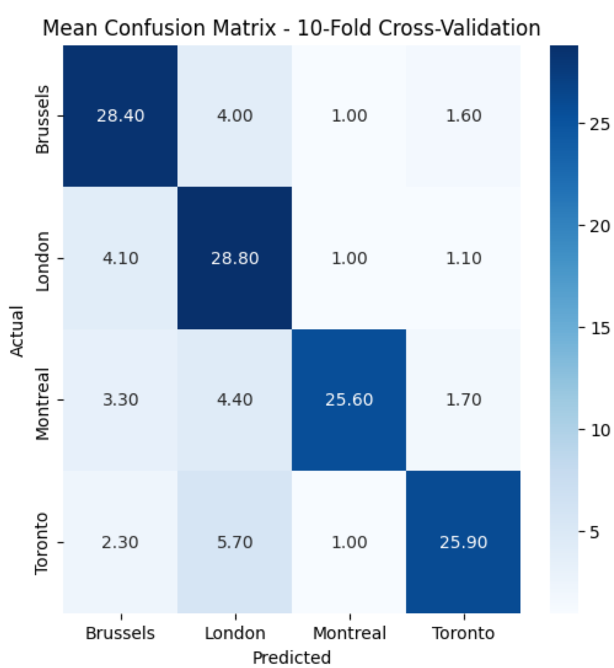


Figure 5: Mean confusion matrix across 10-fold cross validation for model #4 (stacking) (refer to Table 1)

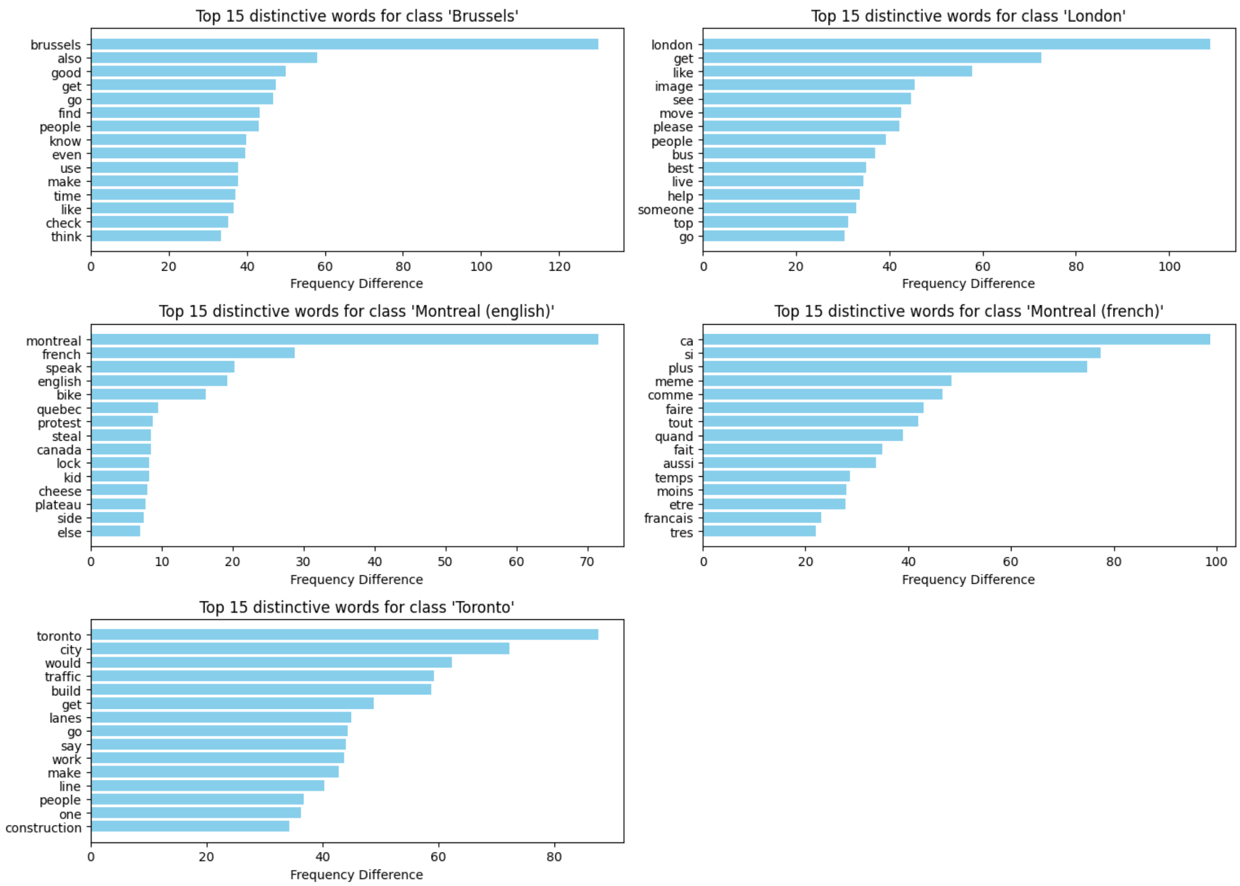


Figure 4: Top 15 most distinctive words per class for the training dataset

Table 2: Hyperparameters selected

Model #	Hyperparameter values
1	k_d = 650, $\alpha = 1$
2	$k_{M.I} = 2850$
3	$k_{M.I} = 3000$, Kernel: Gaussian, $\gamma = 1$
4	**2+3**
5	$\alpha = 1$
7	$C = 1$, Solver: **lbfgs** [5]
8	*itr:* 200, *lr:* 0.05, *d:* 6

Table 3: Models performance results (evaluated on k-fold cross validation)

Model	**Training acc.**	**Validation acc.**	**Time (k-fold) (s)**
Model 1	86.8%	T: 74.12% L: 78.85% B: 80.00% M: 50.90% **Mean: 71.2%**	4.05 (10 - fold)
Model 2	85.9 %	T: 77.14% L: 88.36% B: 81.70% M: 52.23% **Mean: 74.99 %**	3.5 (10 - fold)
Model 3	99 %	T: 67.17% L: 60.74% B: 55.91% M: 90.71% **Mean: 64.55%**	34.7 (10 - fold)
Model 4	93.65 %	T: 85.47% L: 67.39% B: 75.02% M: 90.25% **Mean: 77.77%**	94.81 (10 - fold)
Model 5	60.79%	T: 60.25% L: 58.41% B: 61.80% M: 63% **Mean: 60.865%**	3.2 (5-fold)
Model 6	74%	T: 76.31% L: 72.02% B: 72.55% M: 75.08% **Mean: 73.99%**	1.24 (5-fold)
Model 7	68.71%	T: 72.19% L: 67.62% B: 67.20% M: 67.36% **Mean: 68.59%**	389 (5-fold)

Subreddit prediction

1. Description of the project

Project overview

This project aims to develop machine learning models for **analyzing Reddit text** to determine the origin subreddit of a given post or comment. Reddit, a popular social media platform, is organized into a variety of thematic communities known as *subreddits*, where users share content and engage in discussions.

Objective

The primary objective is to build a model that can **predict the subreddit** of a Reddit post or comment. Given a text entry from Reddit, the model will identify which of the following subreddits it originally came from:

- **Toronto**
- **Brussels**
- **London**
- **Montreal**

This defines a multiclass classification problem

Approach

This project consists of two main parts:

1. **Implement a Bernoulli Naïve Bayes Classifier from Scratch**
   First, a Bernoulli Naïve Bayes classifier will be developed from the ground up, without relying on external libraries for the core algorithm. This implementation will provide a deeper understanding of how the Bernoulli Naïve Bayes method works and how it can be applied to text classification.

2. **Utilize a Classifier from Scikit-Learn**
   In the second part, a pre-built classifier from the `scikit-learn` library will be used to perform the same task. This comparison will allow us to evaluate the effectiveness of our custom implementation against a widely used, optimized machine learning library.

2. Modules importation

Module importation

```
import numpy as np
```

```python
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import PCA
from sklearn.feature_selection import mutual_info_classif
from sklearn.feature_selection import SelectKBest

import time

import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords, words

import langid

# Ensure required NLTK resources are downloaded
try:
    nltk.download('punkt')
    nltk.download('stopwords')
    nltk.download('words')

except Exception as e:
    print(f"Error downloading NLTK resources: {e}")

# Define stopwords list
specific_stopwords = ["https", "subreddit", "www", "com"] ## some
specific words for the given dataset
stopwords_list = stopwords.words('english') +specific_stopwords +
stopwords.words('french') # dataset is both in english and in french
```

```
[nltk_data] Downloading package punkt to /home/clatimie/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/clatimie/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package words to /home/clatimie/nltk_data...
[nltk_data]   Package words is already up-to-date!
```

3. Bernoulli Naïve Bayes Classifier

```python
# Bernoulli Naïve Bayes
class NaiveBayesClassifier:
    def __init__(self, laplace_alpha, unique_labels):
        self.alpha = laplace_alpha  # true for performing Laplace
smoothing
        self.classes = unique_labels
```

```
        self.thetak = None
        self.theta_j_k = None

    def fit(self, X, y):
        # Laplace smoothing parameters
        n_k = self.classes.shape[0]  # number of classes
        n_j = X.shape[1]  # number of features
        n_samples = X.shape[0] # number of samples

        self.theta_k = np.zeros(n_k)  # probability of class k
        self.theta_j_k = np.zeros((n_k, n_j))  # probability of
feature j given class k

        # compute parameters
        for k in range(n_k):
            count_k = (y==self.classes[k]).sum()
            self.theta_k[k] = count_k / n_samples
            for j in range(n_j):
                self.theta_j_k[k][j] = (X[y==self.classes[k], j].sum()
+self.alpha) / (count_k+2*self.alpha)

    def predict(self, X):
        theta_k = self.theta_k  # Prior probabilities P(y)
        theta_j_k = self.theta_j_k  # Conditional probabilities P(X|y)
for each feature and class

        # Calculate log probabilities for P(y) and P(X|y)
        log_theta_k = np.log(theta_k)  # Shape (num_classes,)
        log_theta_j_k = np.log(theta_j_k)  # Shape (num_classes,
num_features)
        log_one_minus_theta_j_k = np.log(1 - theta_j_k)  # Shape
(num_classes, num_features)

        # Calculate the log probabilities of each sample in X for each
class
        probs = (X @ log_theta_j_k.T) + ((1 - X) @
log_one_minus_theta_j_k.T) + log_theta_k

        # Choose the class with the highest probability
        y_pred = np.argmax(probs, axis=1)

        # Transform back to text-based values (class labels)
        return self.classes[y_pred]


    def accu_eval(self, X, y):
        # Predict the classes for the input data
        predicted_classes = self.predict(X)

        # Ensure the predicted classes are in the correct shape
```

```
        # If predicted_classes is already 1D, reshaping is not
necessary
        if predicted_classes.ndim == 1:
            predicted_classes = predicted_classes.reshape((-1, 1))

        # Convert y to a NumPy array if it's a Pandas Series
        if isinstance(y, pd.Series):
            y = y.to_numpy()

        # Calculate accuracy: compare predicted classes with true
labels
        accuracy = np.mean(predicted_classes.flatten() == y.flatten())
        accuracy_per_class = np.zeros((len(self.classes)))


        # Calculate accuracy per class
        for i, cls in enumerate(self.classes):
            # Find indices where the true label is the current class
            class_indices = (y == cls)

            # Calculate the accuracy for the current class
            if np.sum(class_indices) > 0:  # Avoid division by zero
                accuracy_per_class[i] =
np.mean(predicted_classes[class_indices] == y[class_indices])

        return accuracy, accuracy_per_class

    def k_fold_cross_validation(self, k, X, y, print_info=True):
        # Performs k-fold cross-validation to evaluate the model's
performance
        num_samples = X.shape[0]  # Get number of samples in dataset

        indices = np.arange(num_samples)
        np.random.seed(10)
        np.random.shuffle(indices)  # Shuffle the indices
        X = X[indices]  # Apply shuffled indices to X
        y = y[indices]  # Apply shuffled indices to y to maintain
correspondence

        fold_size = num_samples // k  # Calculate size of each fold
        accuracies = []  # Initialize list to store accuracies for
each fold
        accuracies_training = []  # Initialize list for training
accuracies
        accuracies_per_class = []

        for fold in range(k):
            if print_info:
                print(f"\nFold : {fold + 1}")  # Print current fold
number
```

```
            test_start = fold * fold_size  # Start index for test set
            test_end = (fold + 1) * fold_size if fold < k - 1 else
num_samples  # End index for test set

            X_test = X[test_start:test_end, :]  # Create test set
            y_test = y[test_start:test_end]  # Corresponding target
values for test set

            X_train = np.vstack((X[:test_start, :], X[test_end:, :]))
# Create training set
            y_train = np.concatenate((y[:test_start], y[test_end:]))
# Corresponding target values for training set
            if print_info:
                print(f"Class distribution within training dataset :")
# Print class distribution
                for k in range(0, len(self.classes)):
                    print(f'Proportion of class {self.classes[k]} :
{np.sum(y_train==self.classes[k])/len(y_train)*100} %')

            self.fit(X_train, y_train)  # Fit model on training set
            accu_valid, accu_valid_per_class = self.accu_eval(X_test,
y_test) # Evaluate accuracy on test set
            accuracies.append(accu_valid)
            accuracies_per_class.append(accu_valid_per_class)
            accu_training,_ = self.accu_eval(X_train, y_train)
            accuracies_training.append(accu_training)  # Evaluate
accuracy on training set
            if print_info:
                print(f"\n Accuracy = {accuracies[-1]}")  # Print
accuracy for current fold
                print(f"\n Accuracies per class
{accuracies_per_class[-1]}")

        accuracies = np.array(accuracies)  # Convert accuracies list
to NumPy array

        mean_accuracies = np.mean(accuracies)  # Calculate mean
accuracy across folds
        mean_accuracies_training = np.mean(accuracies_training)  #
Calculate mean training accuracy across folds
        std_accuracies = np.std(accuracies)  # Calculate standard
deviation of accuracies
        mean_accu_per_class = np.mean(np.array(accuracies_per_class),
axis=0)

        return mean_accuracies, std_accuracies,
mean_accuracies_training, mean_accu_per_class

    def predict_and_save(self, x, path):
        # Example of how to predict classes
```

```
        predicted_classes = self.predict(x)[:, 0]

        # Create a DataFrame to hold the predictions with an 'id'
column
        df_predictions = pd.DataFrame({
            'id': np.arange(len(predicted_classes)),  # Creates an ID
column starting from 0
            'subreddit': predicted_classes            # Use the
predicted classes as subreddit names
        })

        # Save the DataFrame to a CSV file
        df_predictions.to_csv(path, index=False)
```

4. Lemma and STEM Tokenizer

```
class LemmaTokenizer:
    def __init__(self, stopwords=None):
        self.wnl = WordNetLemmatizer()
        self.stop_words = stopwords

    def __call__(self, doc):
        # Tokenize the document and apply lemmatization and filtering
        return [
            self.wnl.lemmatize(t, pos="v") for t in word_tokenize(doc)
            if t.isalpha() and t.lower() not in self.stop_words]

class StemTokenizer:
    def __init__(self, stop_words=None):
        # Initialize the Porter Stemmer
        self.wnl = nltk.stem.PorterStemmer()
        self.stop_words = stop_words

    def __call__(self, doc):
        # Tokenize the document
        tokens = word_tokenize(doc)
        # Process tokens
        return [self.wnl.stem(t) for t in tokens if t.isalpha() and
t.lower() not in self.stop_words]
```

5. Dataset analysis

Load training dataset

```
np.random.seed(10) # set a random seed to make results reproductible

# Define the path to the training data file
path_training = "../datasets/Train.csv"
```

```
# Read the CSV file into a pandas DataFrame
training_data = pd.read_csv(path_training, delimiter=',')

# Set column names explicitly for better readability
training_data.columns = ['text', 'subreddit']

# Shuffle dataset
training_data = training_data.sample(frac=1,
random_state=42).reset_index(drop=True)

# Separate the training data into two series: texts and subreddit
labels
x_train = training_data['text']          # Contains the Reddit posts
or comments
y_train = training_data['subreddit'] # Contains the subreddit each
post originates from

# Get unique subreddit labels
unique_labels = np.unique(y_train)   # List of unique subreddits in
the dataset

n_samples_training = x_train.shape[0]
n_classes = unique_labels.shape[0]

print(f"Training dataset has {n_samples_training} examples and there
are {n_classes} classes")
```

```
Training dataset has 1399 examples and there are 4 classes
```

Load test dataset

```
# Define the path to the training data file
path_test = "../datasets/Test.csv"

# Read the CSV file into a pandas DataFrame
x_test = pd.read_csv(path_test, delimiter=',')["body"]

n_samples_test = x_test.shape[0]
print(f"Test dataset has {n_samples_test} examples")
```

```
Test dataset has 600 examples
```

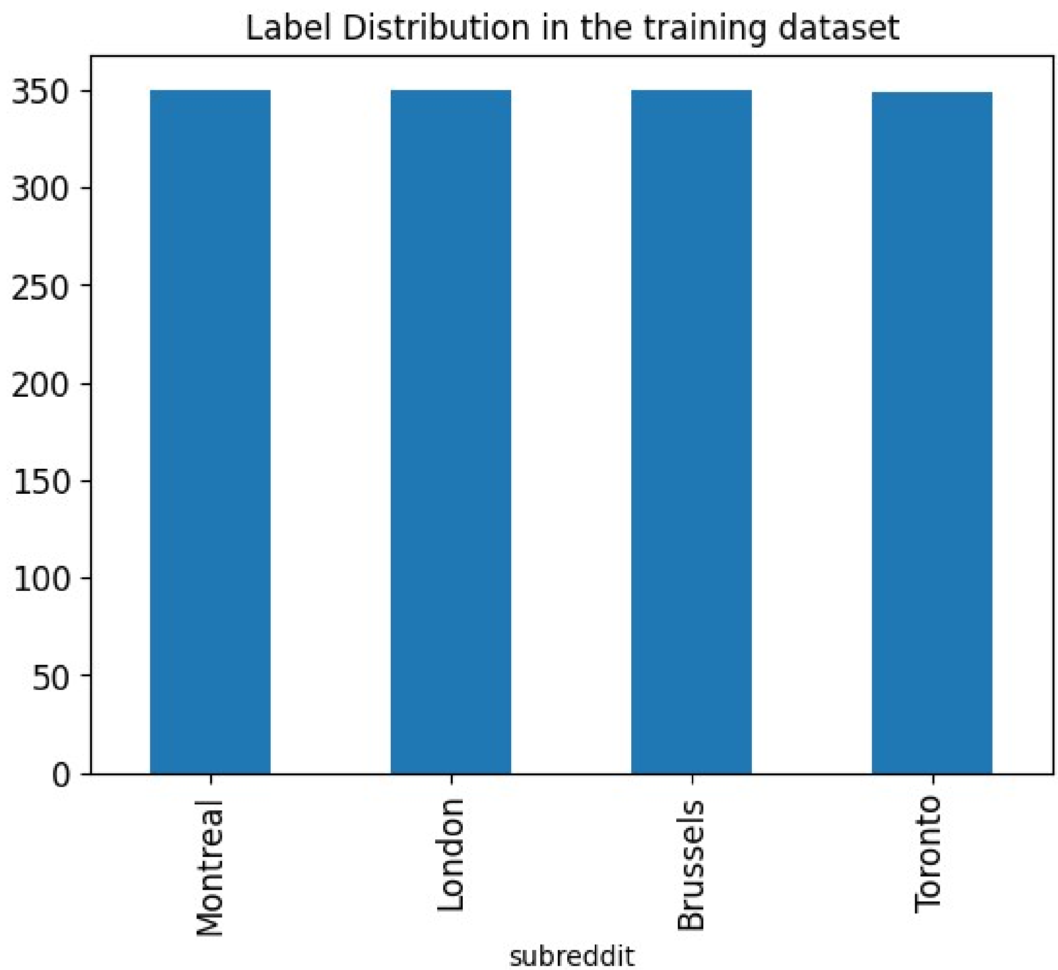
Inspect training dataset

Labels distribution

```
# Show distribution of examples per class
df = pd.DataFrame(training_data)
# Count the number of samples for each label
label_counts = df['subreddit'].value_counts()
```

```
# Plot the distribution
label_counts.plot(kind='bar', title='Label Distribution in the training dataset', fontsize=12)
```

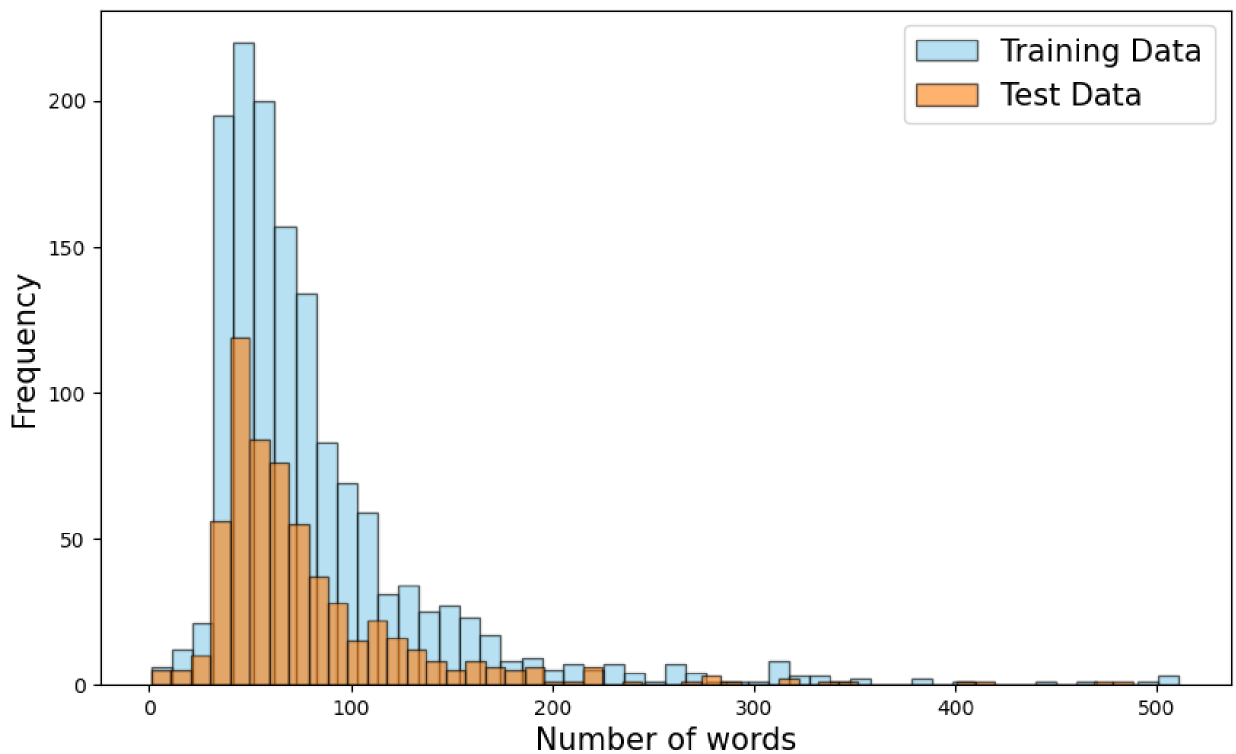
```
<Axes: title={'center': 'Label Distribution in the training dataset'}, xlabel='subreddit'>
```



Text lenght distribution

```
# Calculate the length of each text (in words) for both training and test datasets
text_lengths_train = x_train.apply(lambda x: len(x.split()))
text_lengths_test = x_test.apply(lambda x: len(x.split()))

# Plot both histograms on the same figure
plt.figure(figsize=(10, 6))

# Plot the training dataset histogram
plt.hist(text_lengths_train, bins=50, color='skyblue', edgecolor='black', alpha=0.6, label='Training Data')
```

```
# Plot the test dataset histogram
plt.hist(text_lengths_test, bins=50, color='tab:orange',
edgecolor='black', alpha=0.6, label='Test Data')

# Add labels and title
plt.xlabel('Number of words', fontsize=15)
plt.ylabel('Frequency', fontsize=15)
# Add legend
plt.legend(fontsize=15)

# Show the plot
plt.show()
```



Most distinctive words analysis

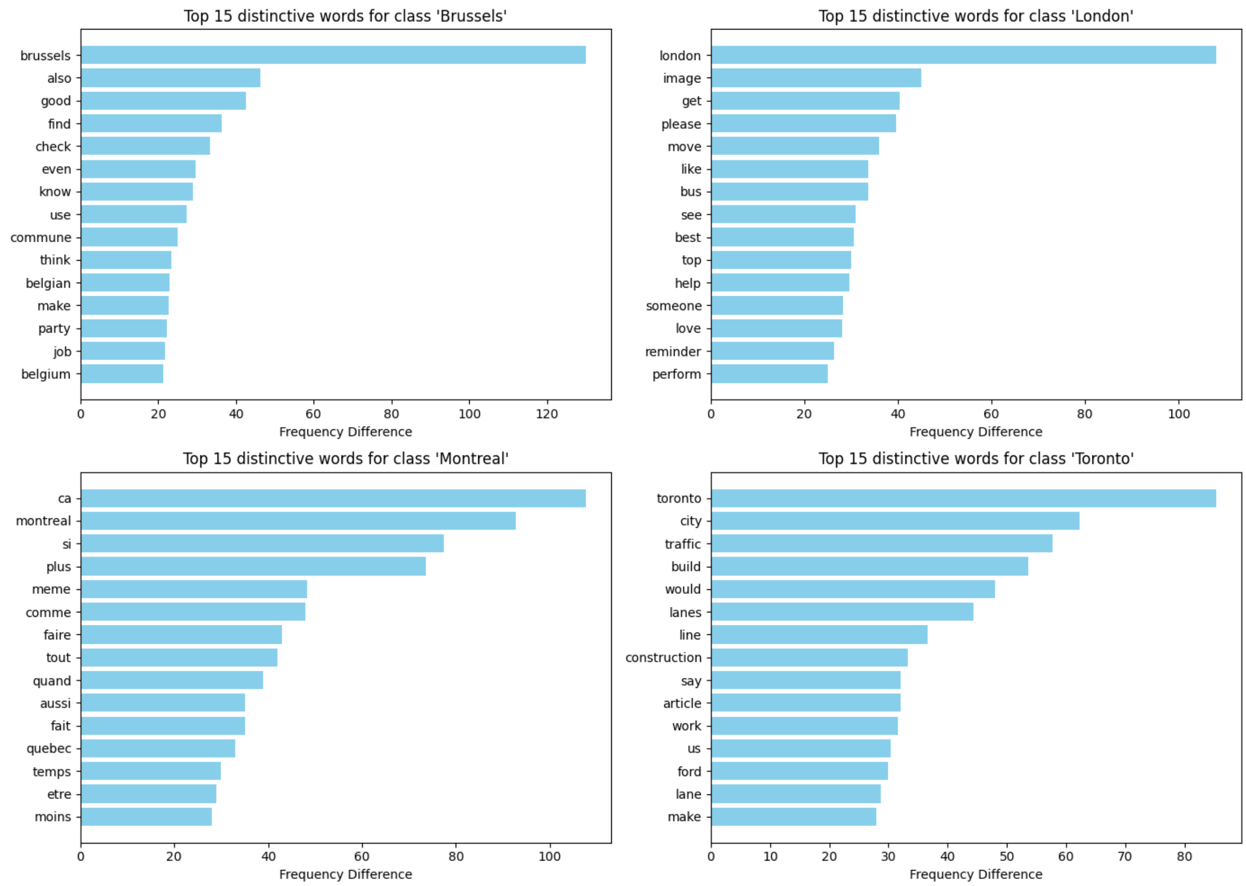
```
def classify_language(comment):
    language, _ = langid.classify(comment)
    return 'Montreal (english)' if language == 'en' else 'Montreal
(french)' if language == 'fr' else 'Montreal (english)'

# Modify the labels for comments in the Montreal class
y_train_mtl_distinct = []  # To hold modified labels

for comment, label in zip(x_train, y_train):
    if label == 'Montreal':
        language = classify_language(comment)
```

```
        y_train_mtl_distinct.append(language)
    else:
        y_train_mtl_distinct.append(label)

def plot_most_distinctive_words_frequency(top_n_plot, texts_train,
y_train, top_n_selected, plots=True):
    unique_labels = sorted(set(y_train))  # Get unique classes
    label_texts = {label: [] for label in unique_labels}  # Dictionary
to hold texts per class

    # Separate texts by label
    for text, label in zip(texts_train, y_train):
        label_texts[label].append(text)

    # Fit CountVectorizer with the custom tokenizer
    vectorizer = CountVectorizer(
        token_pattern=r'\b[a-zA-Z]{2,}\b',
        stop_words=stopwords_list,
        tokenizer=LemmaTokenizer(stopwords=stopwords_list),
        strip_accents="unicode"
    )

    vectorizer.fit(texts_train)
    feature_names = vectorizer.get_feature_names_out()

    # Initialize a dictionary to store word frequencies per class
    word_frequencies = {label: np.zeros(len(feature_names)) for label
in unique_labels}

    # Calculate word frequencies for each word in each class
    for label in unique_labels:
        count_matrix = vectorizer.transform(label_texts[label])
        word_frequencies[label] =
np.array(count_matrix.sum(axis=0)).flatten()

    # List to hold the top distinctive words across all classes
    all_distinctive_words = []

    if plots:
        # Set up the figure with subplots
        n_labels = len(unique_labels)
        n_cols = 2  # Number of columns for subplots
        n_rows = (n_labels + n_cols - 1) // n_cols  # Calculate number
of rows required
        fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, 10))  #
Adjust grid size
        axes = axes.flatten()  # Flatten axes array for easy indexing

    for i, label in enumerate(unique_labels):
        # Calculate distinctiveness by comparing word frequency of
```

```
this class to the average in other classes
        other_classes = [lbl for lbl in unique_labels if lbl != label]

        if label == "montreal_english":
            avg_freq_other_classes =
np.mean([word_frequencies[other_label] for other_label in
other_classes if other_label != "montreal_french"], axis=0)
        elif label == "montreal_french":
            avg_freq_other_classes =
np.mean([word_frequencies[other_label] for other_label in
other_classes if other_label != "montreal_english"], axis=0)
        else:
            avg_freq_other_classes =
np.mean([word_frequencies[other_label] for other_label in
other_classes], axis=0)

        # Calculate distinctiveness score (frequency in this class
minus average frequency in other classes)
        distinctiveness_scores = word_frequencies[label] -
avg_freq_other_classes

        # Get the indices of the top N distinctive words
        if label == "montreal_english" or label == "montreal_french":
            top_n_selected_mtl = int(top_n_selected*0.6)
            top_indices = np.argsort(distinctiveness_scores)[-
top_n_selected_mtl:][::-1]  # Indices of top N scores in descending
order
        else:
            top_indices = np.argsort(distinctiveness_scores)[-
top_n_selected:][::-1]  # Indices of top N scores in descending order

        # Select the top N distinctive words and their scores
        distinctive_words = [feature_names[idx] for idx in
top_indices]
        distinctive_scores = [distinctiveness_scores[idx] for idx in
top_indices]

        # Extend the all_distinctive_words list with the current
class's words
        all_distinctive_words.extend(distinctive_words)

        if plots:
            ax = axes[i]
            ax.barh(distinctive_words[0:top_n_plot],
distinctive_scores[0:top_n_plot], color='skyblue')
            ax.set_xlabel("Frequency Difference")
            ax.set_title(f"Top {top_n_plot} distinctive words for
class '{label}'")
            ax.invert_yaxis()  # Invert y-axis to have the most
distinctive words on top
```

```
            # Adjust layout and show the figure
    if plots:
        for j in range(i + 1, len(axes)):
            axes[j].axis('off')
        plt.tight_layout()
        plt.show()

    # Return the merged list of top distinctive words across all
classes
    return list(set(all_distinctive_words))  # Convert to set to
remove duplicates and back to list


token = plot_most_distinctive_words_frequency(15, x_train, y_train,
top_n_selected=500, plots=True)
```

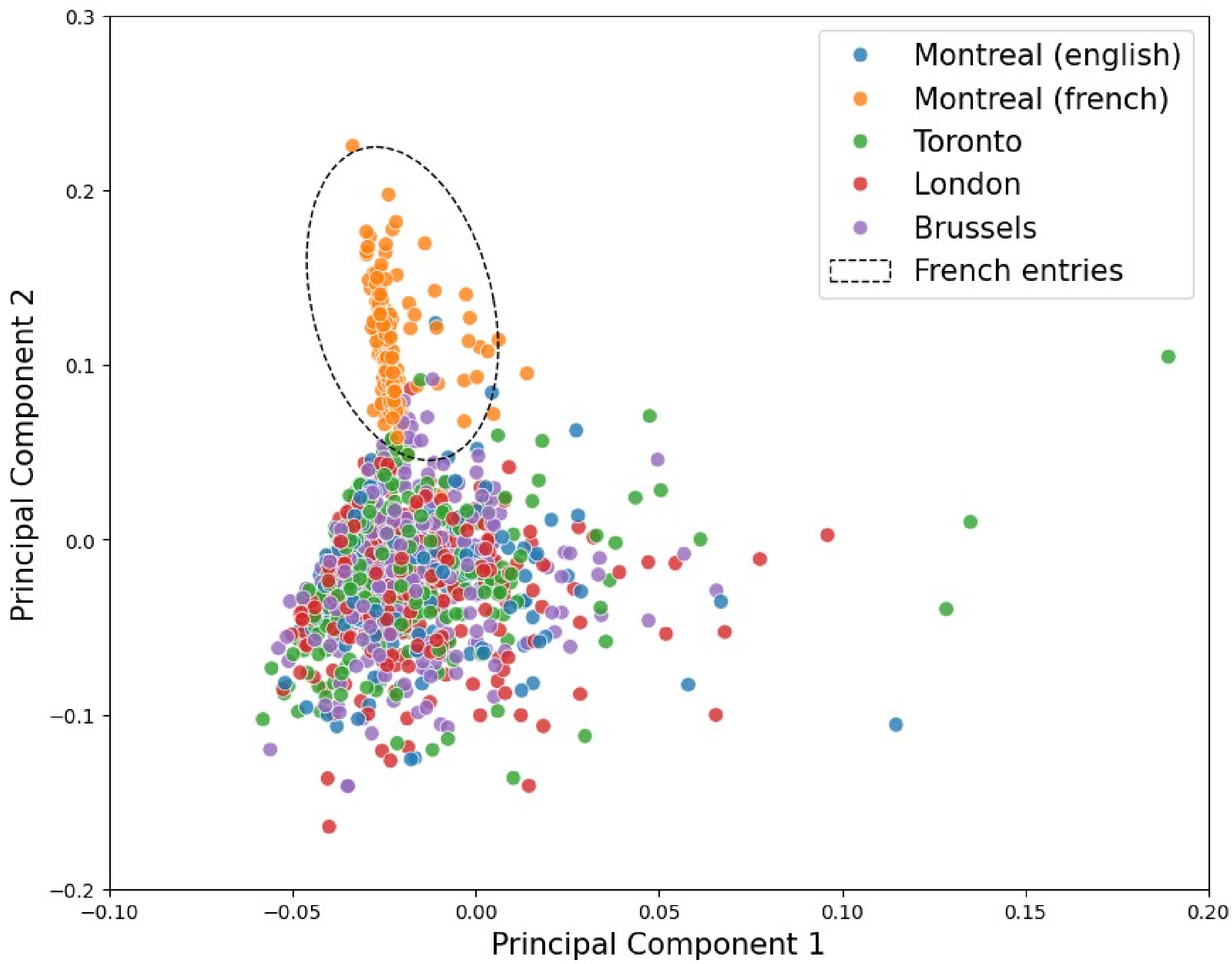
```
/home/clatimie/myenv/lib/python3.12/site-packages/sklearn/
feature_extraction/text.py:521: UserWarning: The parameter
'token_pattern' will not be used since 'tokenizer' is not None'
  warnings.warn(
/home/clatimie/myenv/lib/python3.12/site-packages/sklearn/feature_extr
action/text.py:406: UserWarning: Your stop_words may be inconsistent
with your preprocessing. Tokenizing the stop words generated tokens
['could', 'etaient', 'etais', 'etait', 'etant', 'etante', 'etantes',
'etants', 'ete', 'etee', 'etees', 'etes', 'etiez', 'etions', 'eumes',
'eutes', 'fume', 'futes', 'meme', 'might', 'must', 'need', 'sha',
'wo', 'would'] not in stop_words.
  warnings.warn(
```



PCA Analysis

```
from matplotlib.patches import Ellipse

# PCA Analysis with TF-IDF vectorization
vectorizer = TfidfVectorizer(
    lowercase=True,
    tokenizer=LemmaTokenizer(stopwords=stopwords_list)
)
X_tfidf = vectorizer.fit_transform(x_train)

# Use PCA to reduce dimensionality to 2D
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_tfidf)

# Plot the PCA result with labels
plt.figure(figsize=(10, 8))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1],
hue=y_train_mtl_distinct, palette='tab10', s=60, alpha=0.8)

# Define the ellipse properties
ellipse = Ellipse(
    xy=(-0.02, 0.135),  # Center of the ellipse (mean of the points)
    width=0.18,  # Width of the ellipse
```

```python
    height=0.05,  # Height of the ellipse
    angle=95,  # Rotation angle of the ellipse
    edgecolor='black',  # Color of the ellipse edge
    facecolor='none',  # No fill inside the ellipse
    lw=1,
    linestyle='--',
    label="French entries"
)

# Add the ellipse to the plot
plt.gca().add_patch(ellipse)

# Add titles and labels
plt.xlabel("Principal Component 1", fontsize=15)
plt.xlim(-0.1, 0.2)
plt.ylim(-0.2, 0.3)
plt.ylabel("Principal Component 2", fontsize=15)
plt.legend(loc='best', fontsize=15)
plt.show()
```



6. Vectorization of the Training Texts (BNB)

To utilize the texts in machine learning models, it is essential to convert them into a vectorized format. Below are several methods available for encoding texts as vectors.

Codes

Hyperparameter for BNB

```
def grid_search_naive_bayes_distinctiveness(x_train, y_train,
max_features_list, y_train_mtl, k_cv=10):
    best_accuracy = 0
    best_params = {}
    results = []

    # Iterate over all max_features
    for max_features in (max_features_list):
        print(f"Testing max_features={max_features}")

        vocab =
np.unique(np.array(plot_most_distinctive_words_frequency(20, x_train,
y_train_mtl, top_n_selected=max_features, plots=False)))

        vectorizer = CountVectorizer(
            binary=True, # vectorized vector must be binary for Naive
Bayes
            lowercase=True, # words must be in lowercases
            vocabulary=vocab
        )

        x_train_distinctiveness = vectorizer.fit_transform(x_train)

        classifier = NaiveBayesClassifier(laplace_alpha=1,
unique_labels=unique_labels)
        time_start = time.time()
        mean_accuracy, mean_std, mean_training_accuracy,
mean_accu_per_class = classifier.k_fold_cross_validation(k_cv,
x_train_distinctiveness.todense(), y_train, print_info=False)
        mean_computation_time = 1/k_cv * (time.time() - time_start)

        # Calculate mean accuracy across folds
        results.append((max_features, mean_accuracy, mean_std,
mean_training_accuracy, mean_computation_time, mean_accu_per_class))

        # Update best params if current mean accuracy is the highest
        if mean_accuracy > best_accuracy:
            best_accuracy = mean_accuracy
            best_params = {'max_features': max_features}

    # Output the results of the grid search
```

```
    print("\nGrid search results:")
    for max_features, accuracy, std, mean_training_accuracy,
mean_computation_time, mean_accu_per_class in results:
        print(f"max_features: {max_features} -> Mean Accuracy:
{accuracy:.4f}")
    max_features_values = [result[0] for result in results]
    mean_accuracies = [result[1] for result in results]
    mean_stds = [result[2] for result in results]
    mean_training_accuracies = [result[3] for result in results]
    mean_accu_per_class = np.array([result[5] for result in results])


    # Create a new figure for plotting
    plt.figure(figsize=(10, 6))

    plt.plot(max_features_values, mean_training_accuracies,
label='Training Accuracy', color='g', marker='o', linewidth=2)

    # Add labels and title
    plt.xlabel("Max features per class labels", fontsize=15)
    plt.ylabel("Mean accuracy", color='k', fontsize=15)
    plt.title("Feature selection using distinctiveness scoring")
    plt.legend(loc='upper left')

    # Create a secondary y-axis for validation accuracy
    ax2 = plt.gca().twinx()
    ax2.plot(max_features_values, mean_accuracies, label='Validation
Accuracy', color='b', marker='o', linewidth=2)
    ax2.plot(max_features_values, mean_accu_per_class[:,0],
label='Validation Accuracy - Brussels', color='tab:orange',
marker='+', linestyle='--')
    ax2.plot(max_features_values, mean_accu_per_class[:,1],
label='Validation Accuracy - London', color='tab:red', marker='+',
linestyle='--')
    ax2.plot(max_features_values, mean_accu_per_class[:,2],
label='Validation Accuracy - Montreal', color='tab:purple',
marker='+', linestyle='--')
    ax2.plot(max_features_values, mean_accu_per_class[:,3],
label='Validation Accuracy - Toronto', color='tab:grey', marker='+',
linestyle='--')


    ax2.set_ylabel("Validation Accuracy", fontsize=15)
    ax2.tick_params(axis='y')

    # Show both legends
    ax2.legend(loc='lower right')

    # Show the plot
    plt.show()
```

```
    print(f"\nBest parameter:
max_features={best_params['max_features']} with
accuracy={best_accuracy:.4f}")

    return best_params, best_accuracy


def grid_search_naive_bayes_mutual_information(x_train, y_train,
max_features_list, k_cv=10):
    best_accuracy = 0
    best_params = {}
    results = []

    # Iterate over all max_features
    for max_features in (max_features_list):
        print(f"Testing max_features={max_features}")

        vectorizer = CountVectorizer(
            binary=True, # vectorized vector must be binary for Naive
Bayes
            lowercase=True, # words must be in lowercases
            tokenizer=LemmaTokenizer(stopwords=stopwords_list)
        )

        x_train = vectorizer.fit_transform(x_train)
        x_train_new = SelectKBest(mutual_info_classif,
k=max_features).fit_transform(x_train, y_train)

        classifier = NaiveBayesClassifier(laplace_alpha=1,
unique_labels=unique_labels)
        time_start = time.time()
        mean_accuracy, mean_std, mean_training_accuracy,
mean_accu_per_class = classifier.k_fold_cross_validation(k_cv,
x_train_new.todense(), y_train, print_info=False)
        mean_computation_time = 1/k_cv * (time.time() - time_start)

        # Calculate mean accuracy across folds
        results.append((max_features, mean_accuracy, mean_std,
mean_training_accuracy, mean_computation_time, mean_accu_per_class))

        # Update best params if current mean accuracy is the highest
        if mean_accuracy > best_accuracy:
            best_accuracy = mean_accuracy
            best_params = {'max_features': max_features}

    # Output the results of the grid search
    print("\nGrid search results:")
    for max_features, accuracy, std, mean_training_accuracy,
mean_computation_time, mean_accu_per_class in results:
        print(f"max_features: {max_features} -> Mean Accuracy:
```

```
{accuracy:.4f}")
    max_features_values = [result[0] for result in results]
    mean_accuracies = [result[1] for result in results]
    mean_stds = [result[2] for result in results]
    mean_training_accuracies = [result[3] for result in results]
    mean_accu_per_class = np.array([result[5] for result in results])


    # Create a new figure for plotting
    plt.figure(figsize=(10, 6))

    plt.plot(max_features_values, mean_training_accuracies,
label='Training Accuracy', color='g', marker='o', linewidth=2)

    # Add labels and title
    plt.xlabel("Max features per class labels", fontsize=15)
    plt.ylabel("Mean accuracy", color='k', fontsize=15)
    plt.title("Feature selection using mutual information scoring")
    plt.legend(loc='upper left')

    # Create a secondary y-axis for validation accuracy
    ax2 = plt.gca().twinx()
    ax2.plot(max_features_values, mean_accuracies, label='Validation
Accuracy', color='b', marker='o', linewidth=2)
    ax2.plot(max_features_values, mean_accu_per_class[:,0],
label='Validation Accuracy - Brussels', color='tab:orange',
marker='+', linestyle='--')
    ax2.plot(max_features_values, mean_accu_per_class[:,1],
label='Validation Accuracy - London', color='tab:red', marker='+',
linestyle='--')
    ax2.plot(max_features_values, mean_accu_per_class[:,2],
label='Validation Accuracy - Montreal', color='tab:purple',
marker='+', linestyle='--')
    ax2.plot(max_features_values, mean_accu_per_class[:,3],
label='Validation Accuracy - Toronto', color='tab:grey', marker='+',
linestyle='--')


    ax2.set_ylabel("Validation Accuracy", fontsize=15)
    ax2.tick_params(axis='y')

    # Show both legends
    ax2.legend(loc='lower right')

    # Show the plot
    plt.show()
    print(f"\nBest parameter:
max_features={best_params['max_features']} with
accuracy={best_accuracy:.4f}")
```

```
    return best_params, best_accuracy

#grid_search_naive_bayes_distinctiveness(x_train, y_train,
np.arange(50, 2000, 200), y_train_mtl_distinct, k_cv=10)
#grid_search_naive_bayes_mutual_information(x_train, y_train,
np.arange(50, 4000, 200), k_cv=10)
```

7. K-fold cross validation (BNB + Distinctiveness)

```
k_cv = 10

vocab = np.unique(np.array(plot_most_distinctive_words_frequency(20,
x_train, y_train_mtl_distinct, top_n_selected=650, plots=False)))

vectorizer = CountVectorizer(
    binary=True, # vectorized vector must be binary for Naive Bayes
    lowercase=True, # words must be in lowercases
    vocabulary=vocab
)

x_train_distinctiveness = vectorizer.fit_transform(x_train)
print(f"Feature selection based on distinctiveness ranking: vectorized
training dataset has {x_train_distinctiveness.shape[1]}
tokens/features")


classifier = NaiveBayesClassifier(laplace_alpha=1,
unique_labels=unique_labels)

time_start = time.time()
mean_accuracy, mean_std, mean_training_accuracy, mean_accu_per_class =
classifier.k_fold_cross_validation(k_cv,
x_train_distinctiveness.todense(), y_train, print_info=False)
mean_computation_time = (time.time() - time_start)
print(f'Mean accuracy (training) accross {k_cv}-fold cross
validation : {mean_training_accuracy}')
print(f'Mean variance of validation accuracy accross {k_cv}-fold cross
validation : {mean_std}')
print(f'Mean validation accuracy accross {k_cv}-fold cross
validation : {mean_accuracy}')
print(f'Mean validation accuracy accross {k_cv}-fold cross validation
for class Brussels : {mean_accu_per_class[0]}')
print(f'Mean validation accuracy accross {k_cv}-fold cross validation
for class London : {mean_accu_per_class[1]}')
print(f'Mean validation accuracy accross {k_cv}-fold cross validation
for class Montreal : {mean_accu_per_class[2]}')
print(f'Mean validation accuracy accross {k_cv}-fold cross validation
for class Toronto : {mean_accu_per_class[3]}')
print(f'Computation time  accross {k_cv}-fold cross validation:
{mean_computation_time}')
```

```
/home/clatimie/myenv/lib/python3.12/site-packages/sklearn/
feature_extraction/text.py:521: UserWarning: The parameter
'token_pattern' will not be used since 'tokenizer' is not None'
  warnings.warn(
/home/clatimie/myenv/lib/python3.12/site-packages/sklearn/feature_extr
action/text.py:406: UserWarning: Your stop_words may be inconsistent
with your preprocessing. Tokenizing the stop words generated tokens
['could', 'etaient', 'etais', 'etait', 'etant', 'etante', 'etantes',
'etants', 'ete', 'etee', 'etees', 'etes', 'etiez', 'etions', 'eumes',
'eutes', 'fume', 'futes', 'meme', 'might', 'must', 'need', 'sha',
'wo', 'would'] not in stop_words.
  warnings.warn(
```

```
Feature selection based on distinctiveness ranking: vectorized
training dataset has 2732 tokens/features
Mean accuracy (training) accross 10-fold cross validation :
0.8695141030033117
Mean variance of validation accuracy accross 10-fold cross
validation : 0.04333134284106084
Mean validation accuracy accross 10-fold cross validation :
0.7106455376239549
Mean validation accuracy accross 10-fold cross validation for class
Brussels : 0.8056171622402777
Mean validation accuracy accross 10-fold cross validation for class
London : 0.7796315645274643
Mean validation accuracy accross 10-fold cross validation for class
Montreal : 0.5035571753937008
Mean validation accuracy accross 10-fold cross validation for class
Toronto : 0.7484169322511749
Computation time  accross 10-fold cross validation: 4.5911760330200195
```

8. K-fold cross validation (BNB + Mutual Information)

```
k_cv = 10
vectorizer = CountVectorizer(
            binary=True, # vectorized vector must be binary for Naive
Bayes
            lowercase=True, # words must be in lowercases
            tokenizer=LemmaTokenizer(stopwords=stopwords_list)
        )

x_train = vectorizer.fit_transform(x_train)
selector = SelectKBest(mutual_info_classif, k=2850)
x_train_mi = selector.fit_transform(x_train, y_train)
print(f"Feature selection based on mutual information ranking:
vectorized training dataset has {x_train_mi.shape[1]}
tokens/features")
```

```
classifier = NaiveBayesClassifier(laplace_alpha=1,
unique_labels=unique_labels)

time_start = time.time()
mean_accuracy, mean_std, mean_training_accuracy, mean_accu_per_class =
classifier.k_fold_cross_validation(k_cv, x_train_mi.todense(),
y_train, print_info=False)
mean_computation_time = (time.time() - time_start)
print(f'Mean accuracy (training) accross {k_cv}-fold cross
validation : {mean_training_accuracy}')
print(f'Mean variance of validation accuracy accross {k_cv}-fold cross
validation : {mean_std}')
print(f'Mean validation accuracy accross {k_cv}-fold cross
validation : {mean_accuracy}')
print(f'Mean validation accuracy accross {k_cv}-fold cross validation
for class Brussels : {mean_accu_per_class[0]}')
print(f'Mean validation accuracy accross {k_cv}-fold cross validation
for class London : {mean_accu_per_class[1]}')
print(f'Mean validation accuracy accross {k_cv}-fold cross validation
for class Montreal : {mean_accu_per_class[2]}')
print(f'Mean validation accuracy accross {k_cv}-fold cross validation
for class Toronto : {mean_accu_per_class[3]}')
print(f'Computation time  accross {k_cv}-fold cross validation:
{mean_computation_time}')

classifier.fit(x_train_mi.todense(), y_train)

x_test = vectorizer.transform(x_test)
x_test_mi = selector.transform(x_test)

y_pred = classifier.predict(x_test_mi.todense())

y_pred = y_pred.flatten() if len(y_pred.shape) > 1 else y_pred

# Construct the DataFrame and save to CSV
results_df = pd.DataFrame({
    'id': range(len(y_pred)),
    'subreddit': y_pred
})

# Save predictions to CSV
results_df.to_csv("../output/submissions_mutual_information_bnb.csv",
index=False)
print("Predictions saved to
../output/submissions_mutual_information_bnb.csv")
```

```
/home/clatimie/myenv/lib/python3.12/site-packages/sklearn/
feature_extraction/text.py:521: UserWarning: The parameter
'token_pattern' will not be used since 'tokenizer' is not None'
  warnings.warn(

Feature selection based on mutual information ranking: vectorized
training dataset has 2850 tokens/features
Mean accuracy (training) accross 10-fold cross validation :
0.8590310608655933
Mean variance of validation accuracy accross 10-fold cross
validation : 0.04346068097414368
Mean validation accuracy accross 10-fold cross validation :
0.749863892669648
Mean validation accuracy accross 10-fold cross validation for class
Brussels : 0.8170787233493352
Mean validation accuracy accross 10-fold cross validation for class
London : 0.8835763419696704
Mean validation accuracy accross 10-fold cross validation for class
Montreal : 0.5223190772951375
Mean validation accuracy accross 10-fold cross validation for class
Toronto : 0.7713861288476179
Computation time  accross 10-fold cross validation: 4.608723878860474
Predictions saved to ../output/submissions_mutual_information_bnb.csv
```

File overview

This notebook implements **Support Vector Machines (SVM)** classification for the subreddit prediction dataset. Hyperparameter tuning is performed, and the model's accuracy is evaluated using **10-fold cross-validation**.

Load modules

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import warnings
warnings.filterwarnings("ignore", category=UserWarning)  # This will
suppress UserWarnings



import time

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, classification_report
from sklearn.feature_selection import mutual_info_classif
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline


import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords, words

# Ensure required NLTK resources are downloaded
try:
    nltk.download('punkt')
    nltk.download('stopwords')
    nltk.download('words')

except Exception as e:
    print(f"Error downloading NLTK resources: {e}")

# Define stopwords list
specific_stopwords = ["https", "subreddit", "www", "com"] ## some
specific words for the given dataset
stopwords_list = stopwords.words('english') +specific_stopwords +
stopwords.words('french') # dataset is both in english and in french
```

```
[nltk_data] Downloading package punkt to /home/clatimie/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/clatimie/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package words to /home/clatimie/nltk_data...
[nltk_data]   Package words is already up-to-date!
```

Load training dataset

```
# Define the path to the training data file
path_training = "../datasets/Train.csv"

# Read the CSV file into a pandas DataFrame
training_data = pd.read_csv(path_training, delimiter=',')

# Set column names explicitly for better readability
training_data.columns = ['text', 'subreddit']

# Shuffle dataset
training_data = training_data.sample(frac=1,
random_state=42).reset_index(drop=True)

# Separate the training data into two series: texts and subreddit
labels
x_train = training_data['text']          # Contains the Reddit posts
or comments
y_train = training_data['subreddit'] # Contains the subreddit each
post originates from

# Get unique subreddit labels
unique_labels = np.unique(y_train)   # List of unique subreddits in
the dataset

n_samples_training = x_train.shape[0]
n_classes = unique_labels.shape[0]

print(f"Training dataset has {n_samples_training} examples and there
are {n_classes} classes")
```

```
Training dataset has 1399 examples and there are 4 classes
```

Load test dataset

```
# Define the path to the training data file
path_test = "../datasets/Test.csv"

# Read the CSV file into a pandas DataFrame
x_test = pd.read_csv(path_test, delimiter=',')["body"]
```

```
n_samples_test = x_test.shape[0]
print(f"Test dataset has {n_samples_test} examples")
```

```
Test dataset has 600 examples
```

Lemma Tokenizer from NLTK

```
class LemmaTokenizer:
    def __init__(self, stopwords=None):
        self.wnl = WordNetLemmatizer()
        self.stop_words = stopwords

    def __call__(self, doc):
        # Tokenize the document and apply lemmatization and filtering
        return [
            self.wnl.lemmatize(t, pos="v") for t in word_tokenize(doc)
            if t.isalpha() and t.lower() not in self.stop_words]
```

Hyperparameters search

```
""" # Define the parameter grid for hyperparameter search
param_grid = {
    'svc__kernel': ['linear', 'rbf', 'poly'],  # Different kernel
options
    'select__k': [1000, 2000, 3000, 4000],  # Different values for top
k features
    'svc__C': [0.1, 0.2],  # Different values for C (controls slack in
SVM)
    'svc__gamma': ['scale', 0.001, 0.01, 0.1]  # Gamma values for RBF
and poly kernels
}

# Define the pipeline
pipeline = Pipeline([
    ('vectorizer', TfidfVectorizer(
        lowercase=True,
        tokenizer=LemmaTokenizer(stopwords=stopwords_list)
    )),
    ('select', SelectKBest(mutual_info_classif)),  # Placeholder for k
parameter
    ('scaler', StandardScaler(with_mean=False)),  # Use
with_mean=False for sparse data
    ('svc', SVC())  # SVM classifier
])

# Use GridSearchCV to find the best combination of hyperparameters
grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    cv=5,  # 10-fold cross-validation
```

```
    scoring='accuracy',
    verbose=3,  # To display progress
    n_jobs=-1  # Use all available cores
)

# Fit the model to the training data and search for best parameters
grid_search.fit(x_train, y_train)

# Get the best parameters and corresponding score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best Parameters:", best_params)
print(f"Best Cross-Validated Accuracy: {best_score:.4f}") """
```

```
' # Define the parameter grid for hyperparameter search\nparam_grid =
{\n    \'svc__kernel\': [\'linear\', \'rbf\', \'poly\'],  # Different
kernel options\n    \'select__k\': [1000, 2000, 3000, 4000],  #
Different values for top k features\n    \'svc__C\': [0.1, 0.2],  #
Different values for C (controls slack in SVM)\n    \'svc__gamma\':
[\'scale\', 0.001, 0.01, 0.1]  # Gamma values for RBF and poly
kernels\n}\n\n# Define the pipeline\npipeline = Pipeline([\n
(\'vectorizer\', TfidfVectorizer(\n        lowercase=True,\n
tokenizer=LemmaTokenizer(stopwords=stopwords_list)\n    )),\n
(\'select\', SelectKBest(mutual_info_classif)),  # Placeholder for k
parameter\n    (\'scaler\', StandardScaler(with_mean=False)),  # Use
with_mean=False for sparse data\n    (\'svc\', SVC())  # SVM
classifier\n])\n\n# Use GridSearchCV to find the best combination of
hyperparameters\ngrid_search = GridSearchCV(\n    estimator=pipeline,\
n    param_grid=param_grid,\n    cv=5,  # 10-fold cross-validation\n
scoring=\'accuracy\',\n    verbose=3,  # To display progress\n
n_jobs=-1  # Use all available cores\n)\n\n# Fit the model to the
training data and search for best parameters\ngrid_search.fit(x_train,
y_train)\n\n# Get the best parameters and corresponding score\
nbest_params = grid_search.best_params_\nbest_score =
grid_search.best_score_\n\nprint("Best Parameters:", best_params)\
nprint(f"Best Cross-Validated Accuracy: {best_score:.4f}") '
```

10-fold cross validation

```
vectorizer = TfidfVectorizer(
    lowercase=True,
    tokenizer=LemmaTokenizer(stopwords=stopwords_list)
)

x_train_tfidf = vectorizer.fit_transform(x_train)

selector = SelectKBest(mutual_info_classif, k=3000)
x_train_mi = selector.fit_transform(x_train_tfidf, y_train)
```

```
scaler = StandardScaler()
x_train_svc = scaler.fit_transform(np.asarray(x_train_mi.todense()))

classifier = SVC(kernel="rbf",gamma='scale', C=1)


accuracies = []
class_accuracies = {class_name: [] for class_name in set(y_train)}  #
To store accuracy for each class
kf = KFold(n_splits=10, shuffle=True, random_state=42)
fold = 0

# Start measuring time
start_time = time.time()

accuracies = []
training_accuracies = []
class_accuracies = {class_name: [] for class_name in set(y_train)}  #
To store accuracy for each class
kf = KFold(n_splits=10, shuffle=True, random_state=42)
fold = 0

for train_index, val_index in kf.split(x_train_svc):
    fold += 1
    X_train_fold, X_val_fold = x_train_svc[train_index],
x_train_svc[val_index]
    y_fold_train, y_fold_val = y_train[train_index],
y_train[val_index]

    # Train the classifier
    classifier.fit(X_train_fold, y_fold_train)

    # Predict and evaluate on the validation set
    y_pred = classifier.predict(X_val_fold)
    y_pred_training = classifier.predict(X_train_fold)

    # Display results for each fold
    print(f"\nFold n°{fold}:")

    # Get accuracy per class
    class_accuracy = classification_report(y_fold_val, y_pred,
output_dict=True)
    print("Classification Report:\n",
classification_report(y_fold_val, y_pred))

    accuracy = accuracy_score(y_fold_val, y_pred)
    accuracies.append(accuracy)

    accuracy_training = accuracy_score(y_pred_training, y_fold_train)
    training_accuracies.append(accuracy_training)
```

```python
    for label, metrics in class_accuracy.items():
        if label != 'accuracy' and label!="macro avg" and label!=
"weighted avg":
            class_accuracies[label].append(metrics['precision'])

# Compute total time
end_time = time.time()
total_time = end_time - start_time
print(f"\nTotal computing time for 10 folds: {total_time:.2f}
seconds")

# Mean accuracy across 10 folds
mean_accuracy = np.mean(accuracies)
print(f"Mean Accuracy across 10 folds for SVM classifier:
{mean_accuracy:.4f}")

# Average accuracy for each class
print("\nAverage Accuracy per Class:")
for label, accuracies in class_accuracies.items():
    avg_class_accuracy = np.mean(accuracies)
    print(f"Class {label}: {avg_class_accuracy:.4f}")

# Mean training accuracy across 10 folds
mean_training_accuracy = np.mean(accuracy_training)
print(f"Mean training accuracy across 10 folds for SVM classifier:
{mean_training_accuracy:.4f}")
```

```
Fold n°1:
Classification Report:
              precision    recall  f1-score   support

    Brussels       0.69      0.87      0.77        38
      London       0.71      0.84      0.77        32
    Montreal       0.90      0.59      0.72        32
     Toronto       0.85      0.74      0.79        38

    accuracy                           0.76       140
   macro avg       0.79      0.76      0.76       140
weighted avg       0.79      0.76      0.76       140
```

File overview

This notebook implements a stacking model for subreddit prediction. The stacking classifier combines the predictions of multiple models, including Support Vector Machines (SVM) and Bernoulli Naive Bayes (BNB). Hyperparameter tuning is performed, and the model's performance is evaluated using 10-fold cross-validation.

Load modules

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

import warnings
warnings.filterwarnings("ignore", category=UserWarning)  # This will
suppress UserWarnings

import time

from sklearn.feature_extraction.text import TfidfVectorizer,
CountVectorizer
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.naive_bayes import BernoulliNB
from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import KFold
from sklearn.feature_selection import SelectKBest
from sklearn.metrics import accuracy_score, classification_report
from sklearn.feature_selection import mutual_info_classif
from sklearn.metrics import confusion_matrix

import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords, words

# Ensure required NLTK resources are downloaded
try:
    nltk.download('punkt')
    nltk.download('stopwords')
    nltk.download('words')

except Exception as e:
    print(f"Error downloading NLTK resources: {e}")

# Define stopwords list
specific_stopwords = ["https", "subreddit", "www", "com"] ## some
specific words for the given dataset
```

```
stopwords_list = stopwords.words('english') +specific_stopwords + 
stopwords.words('french') # dataset is both in english and in french
```

```
[nltk_data] Downloading package punkt to /home/clatimie/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/clatimie/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package words to /home/clatimie/nltk_data...
[nltk_data]   Package words is already up-to-date!
```

Load training dataset

```
# Define the path to the training data file
path_training = "../datasets/Train.csv"

# Read the CSV file into a pandas DataFrame
training_data = pd.read_csv(path_training, delimiter=',')

# Set column names explicitly for better readability
training_data.columns = ['text', 'subreddit']

# Shuffle dataset
training_data = training_data.sample(frac=1, 
random_state=42).reset_index(drop=True)

# Separate the training data into two series: texts and subreddit 
labels
x_train = training_data['text']          # Contains the Reddit posts 
or comments
y_train = training_data['subreddit'] # Contains the subreddit each 
post originates from

# Get unique subreddit labels
unique_labels = np.unique(y_train)   # List of unique subreddits in 
the dataset

n_samples_trainings = x_train.shape[0]
n_classes = unique_labels.shape[0]

print(f"Training dataset has {n_samples_training} examples and there 
are {n_classes} classes")
```

```
Training dataset has 1399 examples and there are 4 classes
```

LOad test dataset

```
# Define the path to the training data file
path_test = "../datasets/Test.csv"
```

```python
# Read the CSV file into a pandas DataFrame
x_test = pd.read_csv(path_test, delimiter=',')["body"]

n_samples_test = x_test.shape[0]
print(f"Test dataset has {n_samples_test} examples")
```

```
Test dataset has 600 examples
```

Lemma Tokenizer from NLTK

```python
class LemmaTokenizer:
    def __init__(self, stopwords=None):
        self.wnl = WordNetLemmatizer()
        self.stop_words = stopwords

    def __call__(self, doc):
        # Tokenize the document and apply lemmatization and filtering
        return [
            self.wnl.lemmatize(t, pos="v") for t in word_tokenize(doc)
            if t.isalpha() and t.lower() not in self.stop_words]
```

10 fold cross validation of the stacking model

```python
y_binary = [1 if label == "Montreal" else -1 for label in y_train] #
for svm training

# Define vectorizers
vectorizer_svm = TfidfVectorizer(lowercase=True,
tokenizer=LemmaTokenizer(stopwords=stopwords_list),
strip_accents="unicode")
vectorizer_bnb = CountVectorizer(lowercase=True,
tokenizer=LemmaTokenizer(stopwords=stopwords_list),
strip_accents="unicode")

# Define models
svm_model = SVC(kernel='rbf', probability=True, gamma='scale', C=1)
bnb_model = BernoulliNB()

# Define feature selectors
selector_bnb = SelectKBest(mutual_info_classif, k=2850)
selector_svm = SelectKBest(mutual_info_classif, k=3000)

# Define scaler
scaler_svm = StandardScaler()

# Preprocess data before cross-validation
X_train_bnb = vectorizer_bnb.fit_transform(x_train)
X_train_svm = vectorizer_svm.fit_transform(x_train)

# Apply feature selection
```

```
X_train_bnb_selected = selector_bnb.fit_transform(X_train_bnb,
y_train)
X_train_svm_selected = selector_svm.fit_transform(X_train_svm,
y_binary)

# Scale the SVM features
X_train_svm_scaled =
scaler_svm.fit_transform(np.asarray(X_train_svm_selected.todense()))

# Prepare KFold cross-validation
kf = KFold(n_splits=10, shuffle=True, random_state=42)

accuracies = []
training_accuracies = []
class_accuracies = {class_name: [] for class_name in set(y_train)}  #
To store accuracy for each class

mean_conf_matrix = np.zeros((len(np.unique(y_train)),
len(np.unique(y_train))))  # Initialize empty confusion matrix

fold = 0

# 10-Fold Cross-Validation
time_start = time.time()
for train_index, val_index in kf.split(X_train_svm_scaled):
    fold += 1
    # Split data into training and validation sets
    X_train_fold_svm, X_val_fold_svm =
X_train_svm_scaled[train_index], X_train_svm_scaled[val_index]
    X_train_fold_bnb, X_val_fold_bnb =
X_train_bnb_selected[train_index], X_train_bnb_selected[val_index]
    y_train_bnb_fold, y_val_bnb_fold = np.array(y_train)[train_index],
np.array(y_train)[val_index]
    y_train_svm_fold, y_val_svm_fold = np.array(y_binary)
[train_index], np.array(y_binary)[val_index]

    # Train the models
    svm_model.fit(X_train_fold_svm, y_train_svm_fold)
    bnb_model.fit(X_train_fold_bnb, y_train_bnb_fold)

    # Get predictions from both models
    svm_predictions = svm_model.predict(X_val_fold_svm)
    bnb_predictions = bnb_model.predict(X_val_fold_bnb)

    # Vectorized version of combining predictions
    final_predictions = np.where(svm_predictions == 1, "Montreal",
bnb_predictions)

    # Get predictions from both models for training data
    svm_predictions_training = svm_model.predict(X_train_fold_svm)
```

```
    bnb_predictions_training = bnb_model.predict(X_train_fold_bnb)

    # Vectorized version of combining predictions for training data
    final_predictions_training = np.where(svm_predictions_training ==
1, "Montreal", bnb_predictions_training)

    # Calculate accuracy for this fold
    accuracy = accuracy_score(y_val_bnb_fold, final_predictions)  #
Use y_val_bnb_fold as the correct target variable
    accuracies.append(accuracy)

    training_accuracy  = accuracy_score(y_train_bnb_fold,
final_predictions_training)
    training_accuracies.append(training_accuracy)

    print("Classification Report:\n",
classification_report(y_val_bnb_fold, final_predictions))
    class_accuracy = classification_report(y_val_bnb_fold,
final_predictions, output_dict=True)

    for label, metrics in class_accuracy.items():
        if label != 'accuracy' and label != "macro avg" and label !=
"weighted avg":
            class_accuracies[label].append(metrics['precision'])

    print(f"Validation accuracy for fold {fold}: {accuracy:.4f}")
    print(f"Training accuracy for fold {fold}:
{training_accuracy:.4f}\n")

    # Confusion Matrix for this fold
    conf_matrix = confusion_matrix(y_val_bnb_fold, final_predictions)

    # Add this fold's confusion matrix to the cumulative confusion
matrix
    mean_conf_matrix += conf_matrix
time_end = time.time()
# Calculate the mean confusion matrix
mean_conf_matrix /= kf.get_n_splits()  # Average the confusion matrix

# Plot the mean confusion matrix
plt.figure(figsize=(6, 6))
sns.heatmap(mean_conf_matrix, annot=True, fmt='.2f', cmap='Blues',
xticklabels=np.unique(y_train), yticklabels=np.unique(y_train))
plt.title("Mean Confusion Matrix - 10-Fold Cross-Validation")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()


# Calculate the mean accuracy across all folds
```

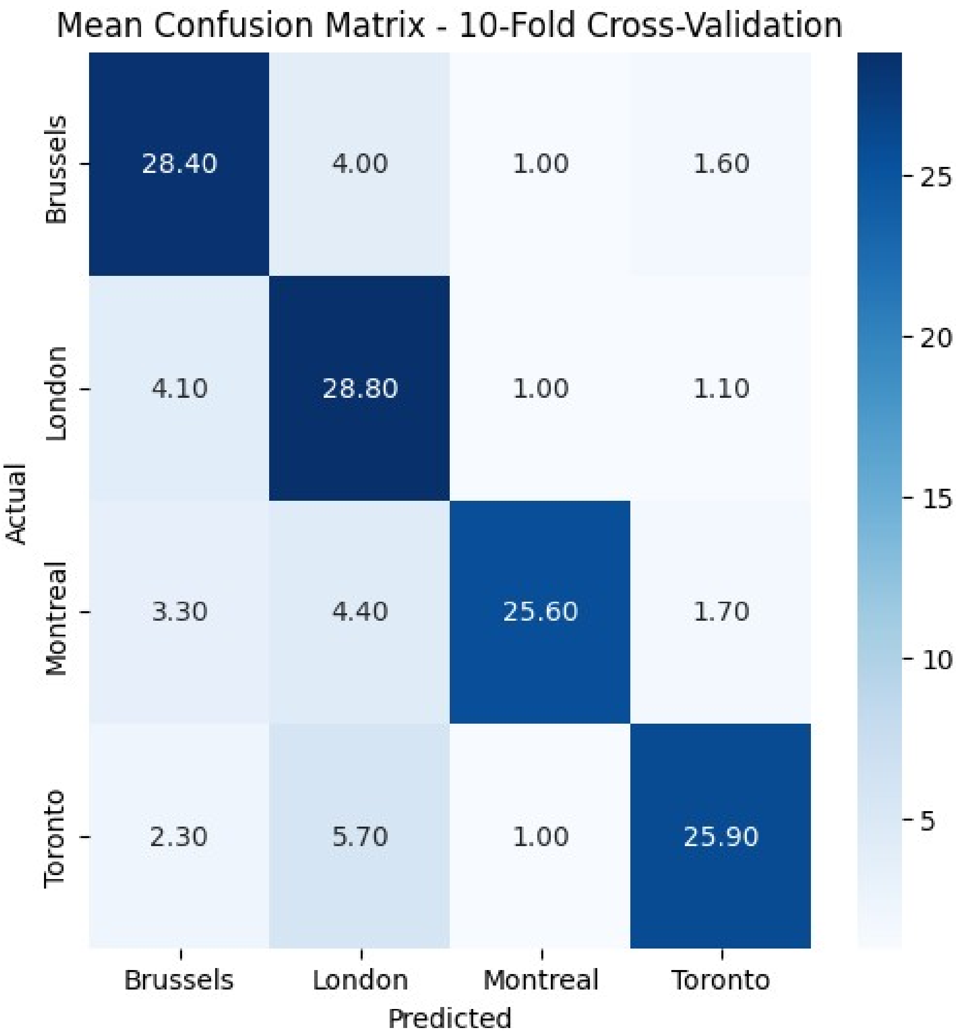
```python
mean_accuracy = np.mean(accuracies)
print(f"Mean Accuracy across 10 folds: {mean_accuracy:.4f}")

mean_training_accuracy = np.mean(training_accuracies)
print(f"Mean Accuracy across 10 folds: {mean_training_accuracy:.4f}")

# Average accuracy for each class
print("\nAverage Accuracy per Class:")
for label, accuracies in class_accuracies.items():
    avg_class_accuracy = np.mean(accuracies)
    print(f"Class {label}: {avg_class_accuracy:.4f}")

print(f"Computing time : {time_end-time_start} (s)")

# Fitting the models with the whole dataset
svm_model.fit(X_train_svm_scaled, y_binary)
bnb_model.fit(X_train_bnb_selected, y_train)

# Preprocess x_test
x_test_bnb = vectorizer_bnb.transform(x_test)
x_test_svm = vectorizer_svm.transform(x_test)

x_test_bnb_selected = selector_bnb.transform(x_test_bnb)
x_test_svm_selected = selector_svm.transform(x_test_svm)

x_test_svm_scaled =
scaler_svm.transform(np.asarray(x_test_svm_selected.todense()))

# Make predictions
svm_predictions = svm_model.predict(x_test_svm_scaled)
bnb_predictions = bnb_model.predict(x_test_bnb_selected)
final_predictions = []
final_predictions = np.where(svm_predictions == 1, "Montreal",
bnb_predictions)


results_df = pd.DataFrame({
    'id': range(len(final_predictions)),
    'subreddit': final_predictions
})

results_df.to_csv("../output/stacking.csv", index=False)
```

```
Classification Report:
               precision    recall  f1-score   support

    Brussels       0.79      0.79      0.79        38
      London       0.62      0.88      0.73        32
    Montreal       0.83      0.75      0.79        32
```

```
     Toronto       0.89      0.66      0.76        38

    accuracy                           0.76       140
   macro avg       0.78      0.77      0.77       140
weighted avg       0.79      0.76      0.77       140

Validation accuracy for fold 1: 0.7643
Training accuracy for fold 1: 0.9333

Classification Report:
              precision    recall  f1-score   support

    Brussels       0.82      0.76      0.79        37
      London       0.62      0.86      0.72        36
    Montreal       1.00      0.82      0.90        28
     Toronto       0.88      0.74      0.81        39

    accuracy                           0.79       140
   macro avg       0.83      0.80      0.80       140
weighted avg       0.82      0.79      0.80       140

Validation accuracy for fold 2: 0.7929
Training accuracy for fold 2: 0.9388

Classification Report:
              precision    recall  f1-score   support

    Brussels       0.86      0.71      0.78        45
      London       0.59      0.87      0.70        30
    Montreal       0.84      0.74      0.79        35
     Toronto       0.82      0.77      0.79        30

    accuracy                           0.76       140
   macro avg       0.78      0.77      0.77       140
weighted avg       0.79      0.76      0.77       140

Validation accuracy for fold 3: 0.7643
Training accuracy for fold 3: 0.9357

Classification Report:
              precision    recall  f1-score   support

    Brussels       0.82      0.88      0.85        42
      London       0.54      0.88      0.67        25
    Montreal       0.96      0.58      0.72        43
     Toronto       0.79      0.73      0.76        30

    accuracy                           0.76       140
   macro avg       0.78      0.77      0.75       140
weighted avg       0.81      0.76      0.76       140
```

```
Validation accuracy for fold 4: 0.7571
Training accuracy for fold 4: 0.9285

Classification Report:
               precision    recall  f1-score   support

    Brussels       0.76      0.81      0.78        31
      London       0.75      0.87      0.80        38
    Montreal       0.96      0.70      0.81        37
     Toronto       0.78      0.82      0.80        34

    accuracy                           0.80       140
   macro avg       0.81      0.80      0.80       140
weighted avg       0.81      0.80      0.80       140

Validation accuracy for fold 5: 0.8000
Training accuracy for fold 5: 0.9444

Classification Report:
               precision    recall  f1-score   support

    Brussels       0.59      0.79      0.68        29
      London       0.76      0.84      0.80        38
    Montreal       0.89      0.68      0.77        37
     Toronto       0.90      0.78      0.84        36

    accuracy                           0.77       140
   macro avg       0.79      0.77      0.77       140
weighted avg       0.80      0.77      0.78       140

Validation accuracy for fold 6: 0.7714
Training accuracy for fold 6: 0.9333

Classification Report:
               precision    recall  f1-score   support

    Brussels       0.82      0.87      0.84        31
      London       0.81      0.83      0.82        36
    Montreal       0.80      0.87      0.84        38
     Toronto       0.86      0.71      0.78        35

    accuracy                           0.82       140
   macro avg       0.82      0.82      0.82       140
weighted avg       0.82      0.82      0.82       140

Validation accuracy for fold 7: 0.8214
Training accuracy for fold 7: 0.9365

Classification Report:
               precision    recall  f1-score   support
```

```
    Brussels       0.61      0.94      0.74        32
      London       0.74      0.66      0.70        44
    Montreal       0.94      0.81      0.87        36
     Toronto       0.86      0.64      0.73        28

    accuracy                           0.76       140
   macro avg       0.79      0.76      0.76       140
weighted avg       0.79      0.76      0.76       140

Validation accuracy for fold 8: 0.7571
Training accuracy for fold 8: 0.9420

Classification Report:
               precision    recall  f1-score   support

    Brussels       0.73      0.77      0.75        35
      London       0.59      0.83      0.69        29
    Montreal       0.88      0.66      0.75        35
     Toronto       0.89      0.78      0.83        41

    accuracy                           0.76       140
   macro avg       0.77      0.76      0.76       140
weighted avg       0.79      0.76      0.76       140

Validation accuracy for fold 9: 0.7571
Training accuracy for fold 9: 0.9380

Classification Report:
               precision    recall  f1-score   support

    Brussels       0.69      0.83      0.76        30
      London       0.72      0.79      0.75        42
    Montreal       0.92      0.76      0.83        29
     Toronto       0.88      0.76      0.82        38

    accuracy                           0.78       139
   macro avg       0.80      0.79      0.79       139
weighted avg       0.80      0.78      0.79       139

Validation accuracy for fold 10: 0.7842
Training accuracy for fold 10: 0.9341
```




```
Mean Accuracy across 10 folds: 0.7770
Mean Accuracy across 10 folds: 0.9365

Average Accuracy per Class:
Class Toronto: 0.8547
Class Montreal: 0.9025
Class London: 0.6739
Class Brussels: 0.7502
Computing time : 105.14465403556824 (s)
```

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from catboost import CatBoostClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import Normalizer, MinMaxScaler
from sentence_transformers import SentenceTransformer

# Load training and test data
train_file = 'Train.csv'
test_file = 'Test.csv'
output_file = 'submissions.csv'

# Training data does not have a header
train_data = pd.read_csv(train_file, header=None, names=['text',
'subreddit'])
test_data = pd.read_csv(test_file)

# Preprocessing metadata (TF-IDF and N-grams)
tfidf_vectorizer = TfidfVectorizer(ngram_range=(1, 2),
max_features=5000)
tfidf_features = tfidf_vectorizer.fit_transform(train_data['text'])

# Perform dimensionality reduction on TF-IDF using TruncatedSVD
svd = TruncatedSVD(n_components=300, random_state=42)
reduced_tfidf = svd.fit_transform(tfidf_features)

# Ensure non-negative features for MultinomialNB
minmax_scaler = MinMaxScaler()
non_negative_tfidf = minmax_scaler.fit_transform(reduced_tfidf)

# Normalize TF-IDF features for Logistic Regression and CatBoost
tfidf_normalizer = Normalizer(norm='l2')
normalized_train_tfidf = tfidf_normalizer.fit_transform(reduced_tfidf)

# Load Sentence Transformer model
sentence_model = SentenceTransformer('paraphrase-multilingual-MiniLM-
L12-v2')
sentence_embeddings =
sentence_model.encode(train_data['text'].tolist())

# Normalize Sentence Embeddings
sentence_normalizer = Normalizer(norm='l2')
normalized_train_sentences =
sentence_normalizer.fit_transform(sentence_embeddings)

# Combine features (Normalized TF-IDF + Normalized Sentence
```

```
Embeddings)
X_combined = np.hstack([
    normalized_train_tfidf,             # Normalized TF-IDF features
(300 dims)
    normalized_train_sentences          # Normalized Sentence
Embeddings (84 dims)
])

# Map labels
label_map = {label: idx for idx, label in
enumerate(train_data['subreddit'].unique())}
y = train_data['subreddit'].map(label_map)

# Hyperparameter grids
param_grid_nb = {
    'alpha': [0.01, 0.1, 1.0]
}

param_grid_lr = {
    'C': [0.1, 1, 10],
    'solver': ['lbfgs']
}

param_grid_cb = {
    'iterations': [100, 200],
    'learning_rate': [0.05],
    'depth': [4, 6]
}

# Multinomial Naive Bayes
nb_model = MultinomialNB()
grid_search_nb = GridSearchCV(
    nb_model, param_grid=param_grid_nb, cv=3, scoring='accuracy',
verbose=1, n_jobs=-1
)
grid_search_nb.fit(non_negative_tfidf, y)

# Logistic Regression
lr_model = LogisticRegression(max_iter=1000)
grid_search_lr = GridSearchCV(
    lr_model, param_grid=param_grid_lr, cv=3, scoring='accuracy',
verbose=1, n_jobs=-1
)
grid_search_lr.fit(X_combined, y)

# CatBoost
cb_model = CatBoostClassifier(verbose=0)
grid_search_cb = GridSearchCV(
    cb_model, param_grid=param_grid_cb, cv=3, scoring='accuracy',
verbose=1, n_jobs=-1
```

```
)
grid_search_cb.fit(X_combined, y)

# Save best estimators
nb_best_model = grid_search_nb.best_estimator_
lr_best_model = grid_search_lr.best_estimator_
cb_best_model = grid_search_cb.best_estimator_

# Metrics
metrics = {
    'Classifier': ['Multinomial Naive Bayes', 'Logistic Regression',
'CatBoost'],
    'Training Accuracy': [
        grid_search_nb.best_score_,
        grid_search_lr.best_score_,
        grid_search_cb.best_score_
    ]
}
metrics_df = pd.DataFrame(metrics)
print(metrics_df)

# Process test set: TF-IDF
test_tfidf_features = tfidf_vectorizer.transform(test_data['body'])
test_reduced_tfidf = svd.transform(test_tfidf_features)  # Reduce
dimensions to 300
test_normalized_tfidf = tfidf_normalizer.transform(test_reduced_tfidf)
# Normalize TF-IDF

# Process test set: Sentence Embeddings
test_sentence_embeddings =
sentence_model.encode(test_data['body'].tolist())
test_normalized_sentence_embeddings =
sentence_normalizer.transform(test_sentence_embeddings)  # Normalize
Sentence Embeddings

# Combine test features (TF-IDF + Sentence Embeddings)
test_combined = np.hstack([
    test_normalized_tfidf,              # Normalized TF-IDF features
(300 dims)
    test_normalized_sentence_embeddings  # Normalized Sentence
Embeddings (84 dims)
])

# Predict on test set using best CatBoost model
test_predictions = cb_best_model.predict(test_combined)

# Map predictions back to labels
reverse_label_map = {idx: label for label, idx in label_map.items()}

# Flatten predictions and map back to labels
```

```
test_predictions = test_predictions.flatten() if
len(test_predictions.shape) > 1 else test_predictions
test_data['subreddit'] = [reverse_label_map[int(pred)] for pred in
test_predictions]

# Create submission file
submission = test_data[['id', 'subreddit']]
submission.to_csv(output_file, index=False)
print(f"Submission file saved as: {output_file}")
```

```
Fitting 3 folds for each of 3 candidates, totalling 9 fits
Fitting 3 folds for each of 3 candidates, totalling 9 fits
Fitting 3 folds for each of 4 candidates, totalling 12 fits
                Classifier  Training Accuracy
0  Multinomial Naive Bayes           0.589274
1      Logistic Regression           0.731398
2                 CatBoost           0.684257
Submission file saved as: submissions.csv
```