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
```

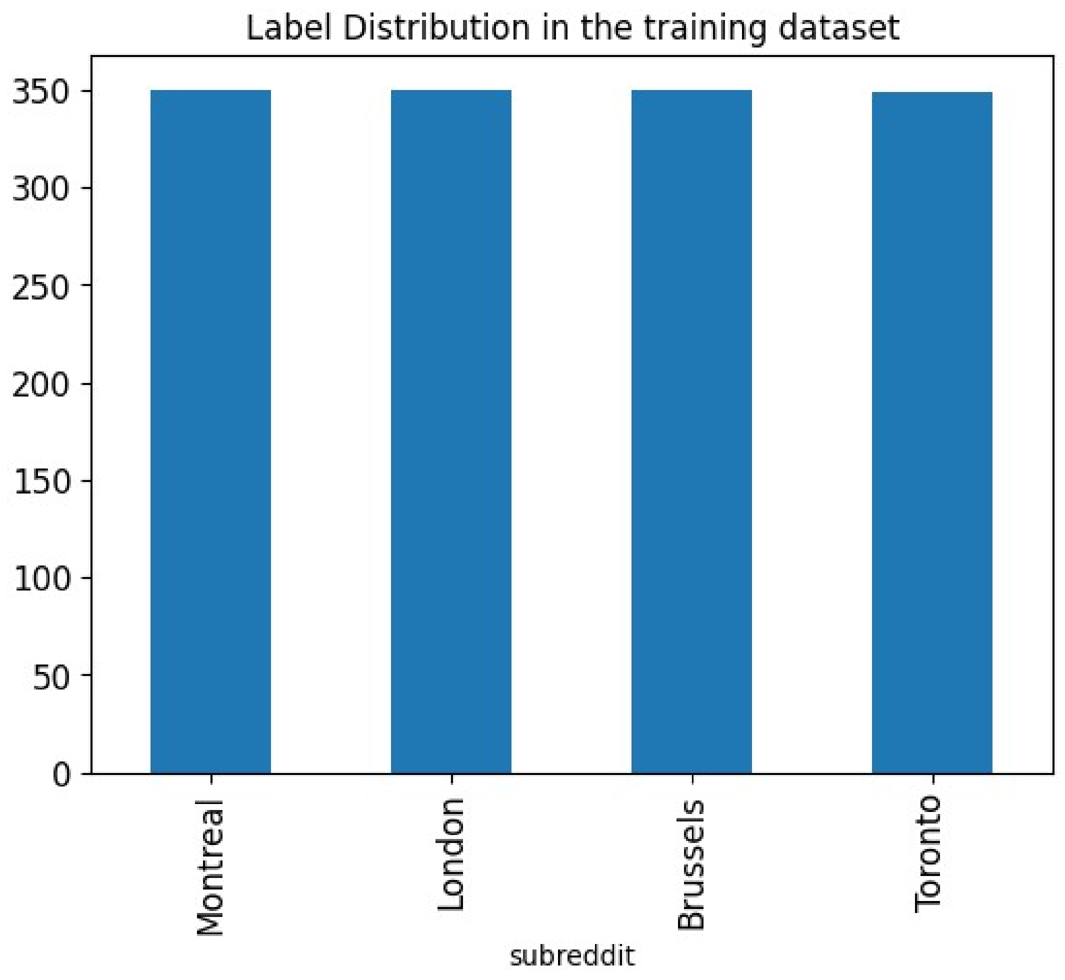
Inspect training dataset

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()
```

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

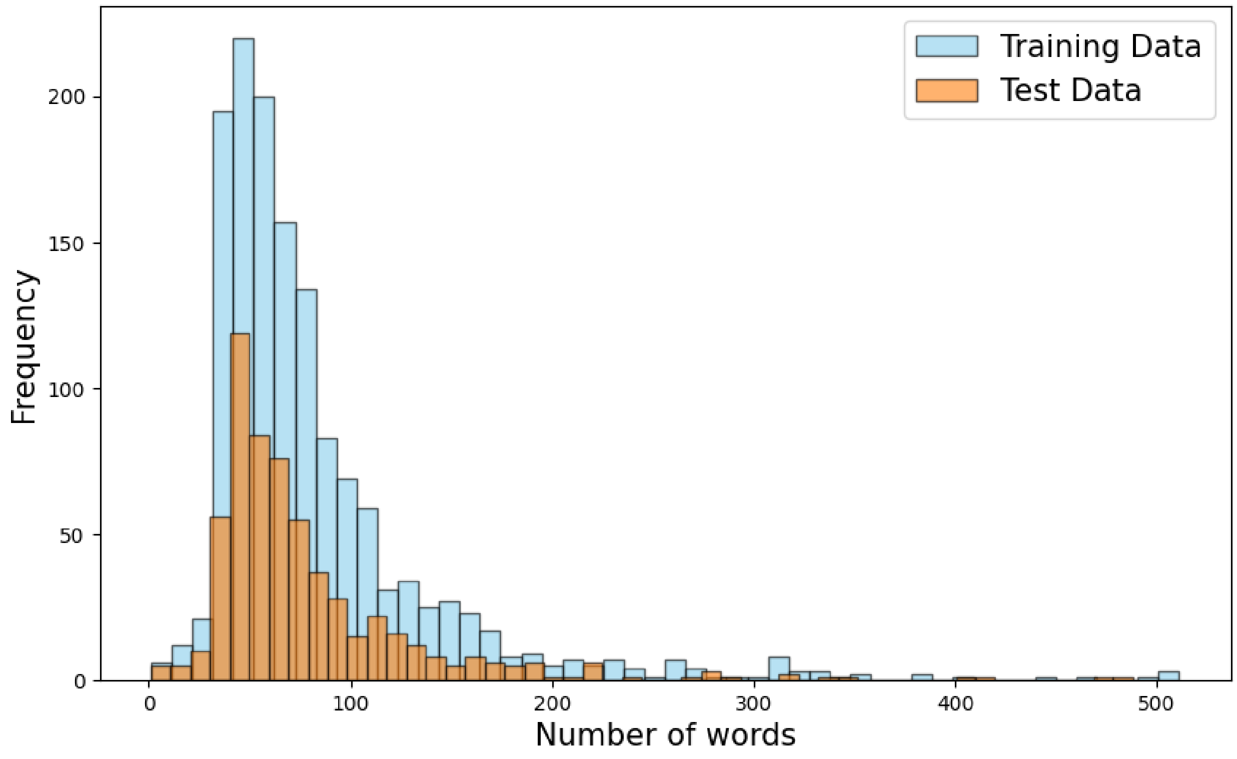
```
<Axes: title={'center': 'Label Distribution in the training dataset'},
xlabel='subreddit'>
```



Text lenght distribution

```python
# 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')
```

```python
# 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()
```



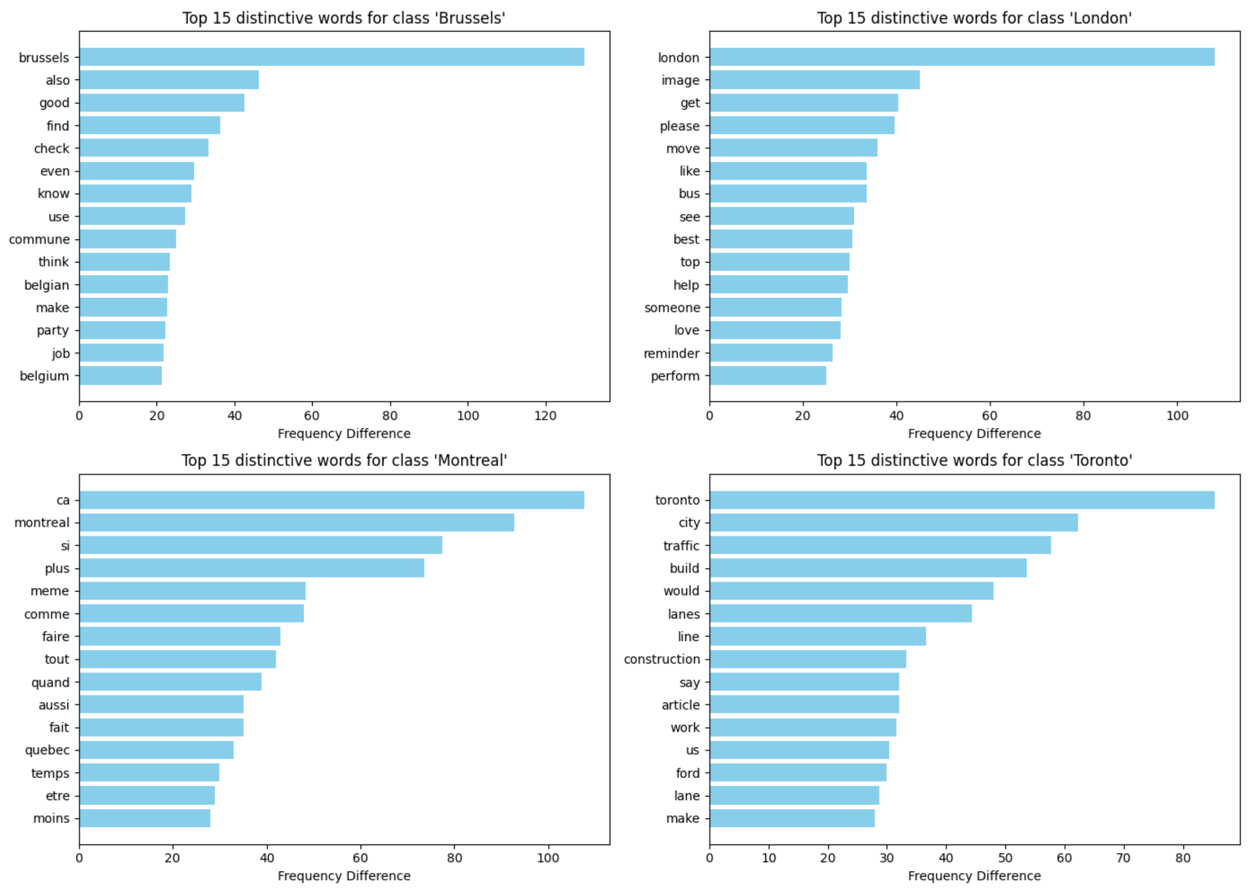
Most distinctive words analysis

```python
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
```

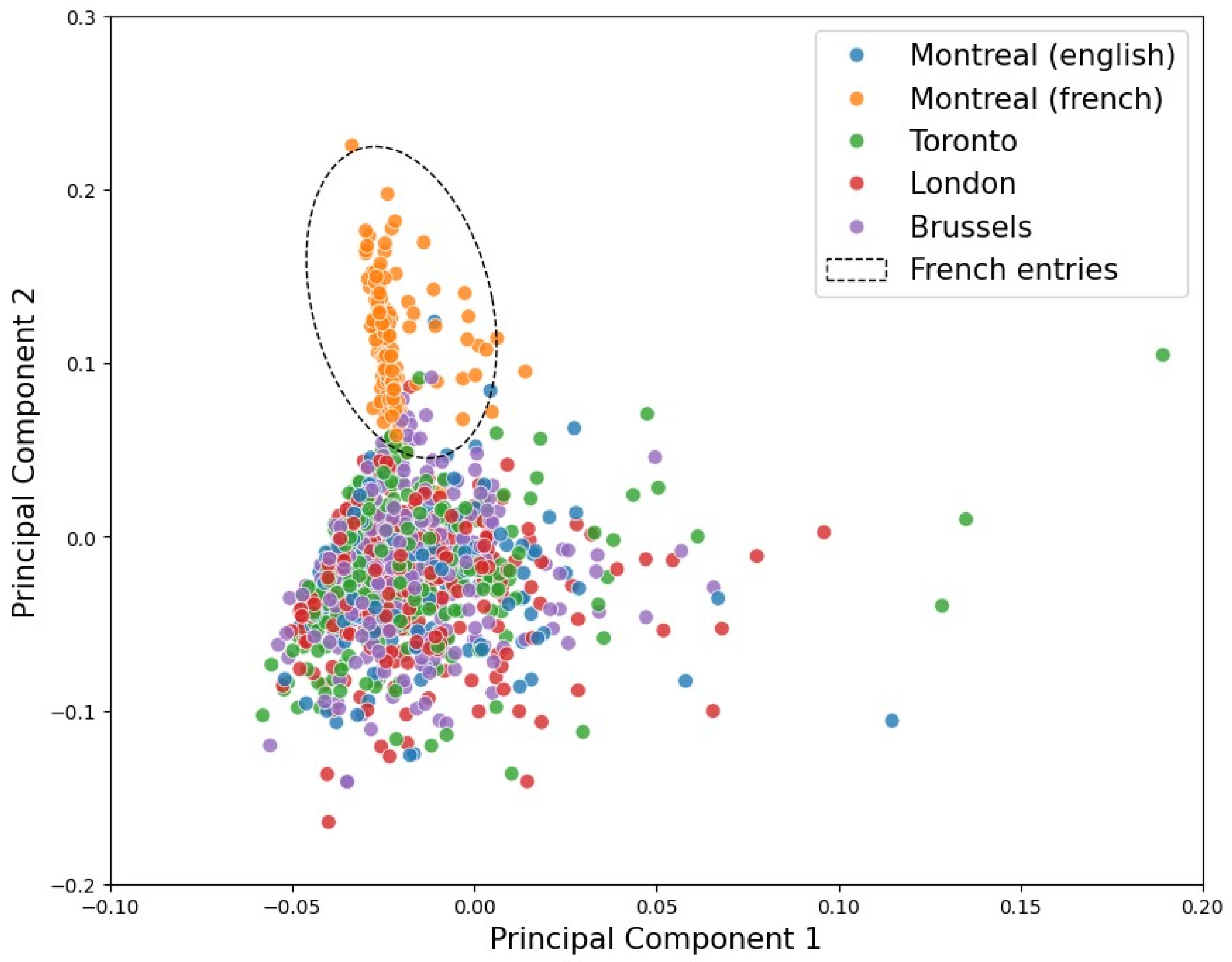
```
            # Adjust layout and show the figure
    if plots:
        for j in range(i + 1, len(axes)):
            axes[j].axis('off')
        plt.tight_layout()
        plt.show()

    # 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
    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:
```

```python
{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")
```

```python
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)
    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

```python
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
```

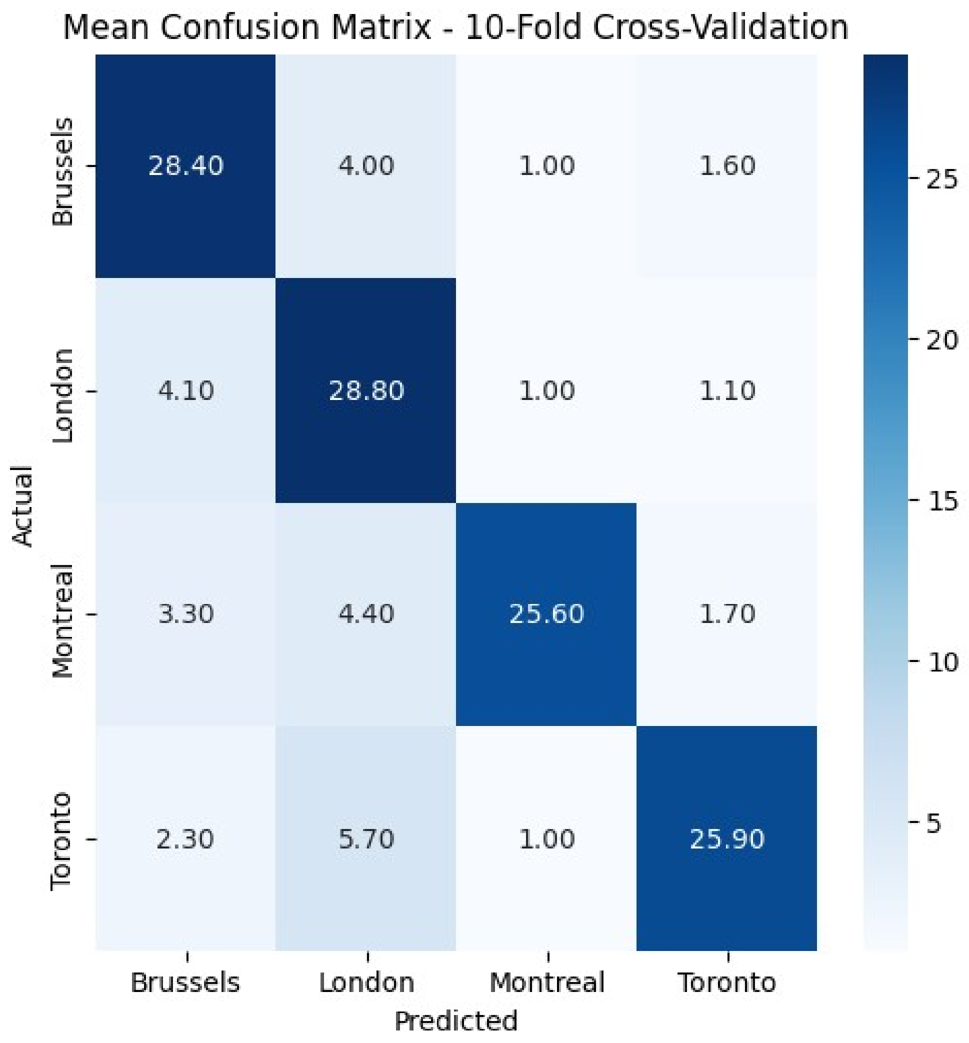
```
X_train_bnb_selected = selector_bnb.fit_transform(X_train_bnb,
y_train)
X_train_svm_selected = selector_svm.fit_transform(X_train_svm,
y_binary)

# 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
```

```
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
```

```python
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
```