ECSE 551 A2: Bernoulli Naïve Bayes Classification

Ashley Meagher (260822930), Charles Sirois (261158513)

Abstract

This project classifies posts on Reddit into four subreddits, London, Montreal, Paris, and Toronto, using the Bernoulli Naïve Bayes classification. The project requires text classification to take the text from the posts as input and assign it to a label (the subreddit). In the project, the first dataset contains posts for training and the second dataset contains posts to test. To improve the accuracy of the text classification, the features are preprocessed using stop-words, lemmatization, n-grams, language detection, and feature selection. Scikit-learn Decision Tree and SVM algorithms are used to compare the classification method, specifically using k-fold cross-validation. After feature manipulation and through cross-validation, it was found that the SVM classifier had the best results of all classifiers, with an accuracy of 72.2% and performance accuracy of 71.1% in the Kaggle competition.

1 Introduction

The goal of this project is to gain experience working with text classification in machine learning. Text classification is an important tool that filters text data, used in many real-life applications such as phishing emails and Google searches. This project implements the Bernoulli Naïve Bayes classifier to filter Reddit posts. Reddit is a social media form where users comment on content associated with different subreddits. For this project, each comment is associated with one of four subreddits: London, Montreal, Paris, and Toronto. The performance and accuracy were then compared to the Decision Tree and Support Vector Machines (SVM) methods using k-fold validation.

1.1 Classifiers

Decision Tree classification is a machine learning tool that uses values of each feature to recursively split the dataset to a point where all data points for the same class are grouped. SVM is another classifier that separates the training data by maximizing the margins between labels [1]. The python package *scikits.learn* was used to implement the Decision Tree and SVM classification. These two machine learning tools were selected since they are known to work well with text classification, which is compared to test the performance of the Bernoulli Naïve Bayes classifier. The hyper-parameters for the Decision Tree and SVM classification were selected for the highest k-fold accuracy but were not as good as the Bernoulli Naïve Bayes classifier.

There are no hyper-parameters associated with the Naïve Bayes classifier [2], however, there are preprocessing strategies applied to the features to ensure the best possible accuracy. Some of these strategies include implementing stop-words, lemmatization, n-grams, and language detection. Bernoulli Naïve Bayes only accepts binary data, which gave an accuracy of 77.5% even after preprocessing the data. Other Naïve Bayes approaches were implemented, including Laplace smoothing. By including these methods, acceptable accuracy was achieved.

1.2 Assumptions

Assumptions are required to properly build a model and train the data efficiently. The main assumption for this project was that the features, x_j are conditionally independent given the label y. This means that it is assumed that a feature presented in a class is unrelated to any other feature [3]. In mathematical terms,

$$P(x_j|y) = P(x_j|y, x_k). \tag{1}$$

2 Dataset Analysis

The datasets given for the project are user text from the Reddit website. The text is associated with one of four subreddits: London, Montreal, Paris, or Toronto. The features represent the words available in the text, while the labels are the associated subreddit. Two datasets are provided, one containing training data and one with test data. The training set has 718 samples that are relatively equally distributed amongst the four classes, and the test set has 279 samples.

2.1 Vectorization

CountVectorizer is a function within the sklearn Python library that tokenizes data. This package was used to split the words of the Reddit comments, however, some of the default settings were modified. The modifications were the removal of all accents, numbers, and punctuation. All words were also converted to lowercase, to ensure the same words are only accounted for once. The samples were then turned into binary features, where 1 represents the existence of a word and 0 represents it is non-existent in the sample. This method is effective for showing the presence of a word but lacks the ability to show its frequency in the sample.

2.2 Language Detection

Since the primary language in Toronto and London is English, while the primary language in Montreal and Paris is French, the language of the Reddit post was considered for the project. The langdetect package in Python is able to identify the language used in the post [4]. Two additional features were added to each post, one identifying the language as English, and the other as French.

2.3 Stop Words

Stop-words are common words that are likely to appear in multiple subreddits, despite their label. These words do not add any value to classifying the comments and therefore are removed from the feature list. In Python, there exist packages for common words in both English and French. Some examples of these words for both languages would include "and", "the", "he", "et", "le", and "lui". These words were removed from the text classification search, therefore putting more emphasis on more important words that will help properly identify the comment's label.

2.4 Lemmatization and Stemmatization

Languages consist of several words that are derived from each other, for example, "change", "changing", and "changed". Lemmatization and stemmatization are techniques that group together different forms of the same word. By combining the words together, the number of features is reduced, and a more efficient analysis is completed. Lemmatization keeps the root word of a similar word, which is more accurate for text classification. Stemmatization only takes the root of a word, without considering if the root is the readable word. For the previous example, lemmatization would group the words together as "change", while stemmatization would group them together as "chang".

2.5 N-Grams

N-grams is a method that uses a series of words instead of each individual word. The sequence of words will have a length n, which was limited in length to prevent a series of too many words. N-grams are important because some words could appear in the same order in multiple samples. For example, if you consider the trigram "The Three Musketeers", these words would often appear in series which could be useful, whereas the term "three" alone may not be useful for classification.

2.6 Feature Selection

The number of features is reduced to the most frequent words observed throughout all posts. This is an important step because keeping too many features would result in over-fitting, but not enough features would result in insufficient classification. To reduce the number of features, the vector containing the binary values is reduced to the most frequent words observed throughout all posts. The number of words to keep was altered and validated using k-fold validation. This process is done using the 'feature_selection' class from *sklearn*. It selects the best n features by comparing their ANOVA F-value [5].

3 Proposed Approach

3.1 Considered Models

3.1.1 Bernoulli Naïve Bayes Classification

Naïve Bayes is a common text classification method because it is fast and easy to implement [3]. It is a probabilistic classifier, which means when given an input it will predict the probability of each class occurring. The Bernoulli Naïve Bayes classifier is a variant of the standard Naïve Bayes, where only discrete data is used and the features are provided in binary form. The assumptions for the classifier are specified in Section 1.2. The probability, $P(y = k|x)$, with k distinct output values is calculated for each class and the class with the highest probability is selected. The formula used to find the probability is

$$P(y = k|x) \propto \delta_k(x) = \log P(y = k)P(x|y = k) \tag{2}$$

$$= \log \theta_k + \sum_{j=1}^{m} (x_j * \log \theta_{j,k} + (1 - x_j) * \log(1 - \theta_{j,k})). \tag{3}$$

From Laplace smoothing,

$$\theta_k = P(y = k) = \frac{\text{number of instances where}(y = k) + 1}{\text{total number of samples} + 2} \tag{4}$$

$$\theta_{j,k} = P(x_j = 1|y = k) = \frac{\text{number of instances where}(x_j = 1)\text{and}(y = k) + 1}{\text{total number of samples with}(y = k) + 2}. \tag{5}$$

The following decision rule assigns the class,

$$Output = \arg\max_x \delta_k(x). \tag{6}$$

3.1.2 Decision Tree

The Decision Tree classification was selected from the *scikit.learn* package to classify the data. Decision Trees classify the data by recursively splitting the data into different features. This process is repeated until all data points for the same class are grouped. The Decision Tree classifier was selected as an additional classifier because it requires little data preprocessing and is simple to implement. Three hyperparameters are considered:

- Tree depth: Maximum depth of the tree. Useful to reduce over-fitting. (Values considered: 50, 100, 500, 1000)
- Minimum sample split: Minimum number of samples in a node to split (Values considered: 2, 5, 10)
- Criterion: Which function to use to measure the quality of a feature. Two options are tested: *entropy* and *gini*.

3.1.3 SVM Classifier

A second classification method was selected to better compare the results. SVM classifier was selected as one of the additional *scikit.learn* packages. As discussed in Section 1, SVM classifier is a supervised learning method that classifies the data by maximizing the margins between labels. SVM was selected because it works well for high-dimensional spaces, which is relevant to text classification since each word represents a feature [6]. SVM classification is a common machine-learning method for text classification. The strength of regularization is the only hyperparameter considered. Values from 1 to $1e - 4$ will be tested. Lower values mean a stronger regularization and thus, less over-fitting.

3.2 Model Selection

The best model will be selected using their K-fold cross-validation accuracy and bias. Since a training and test dataset are provided, all the 718 training samples are used in training. K-fold cross-validation divides the dataset into k subsets, also known as folds. The data is trained and evaluated k times, using a different validation set each time. The average prediction accuracy over all k folds is computed. The model is then trained on the full dataset and its accuracy is measured. The difference between this accuracy and 100% is known as the bias.

To test all possible combinations of hyper-parameters with all the pre-processing options, an automated loop was implemented.

4 Results

The best models obtained for each classification method are presented in Table 1. To find the best combinations of hyper-parameters and pre-processing, an iterative process was used. The quality of the models was compared based on their 5-fold cross-validation accuracy and bias (1 - *training accuracy*).

First, the different pre-processing approaches were tested on the Bernoulli Naïve Bayes classifier. The best feature space for this model was found by following these steps:

1. Vary the number of features
2. Test different N-grams
3. Add language identification to features
4. Lemmatize the vocabulary

At each step, the best parameters are selected and kept constant for the next step. The best combinations of pre-processing were then used to train the other two models: SVM and decision trees. Note that when some combinations of pre-processing gave similar accuracies for Naïve Bayes, both were tested.

For SVM, a linear kernel with regularization was used. The impact of the regularization strength was analyzed. The effects of the criterion, tree depth, and minimum samples to split were tested for decision trees. For both models, the count of words was used instead of their binary value.

Also, since lemmatization did not improve the accuracy, and lemmatization of the dataset is computationally demanding, this option was not tested on the other model.

Table 1: Summarized results of the iterative process to find the best model

(a) Bernouilli Naïve Bayes

Test	CV [%]	Bias [%]
Step 1 — N Features		
500	69.7	22.81
1000	71.5	18.36
2000	**74.8**	**11.54**
3000	67.5	13.21
4000	62.9	13.21
Step 2 — N-Grams		
2-grams	**77.5**	**11.8**
3-grams	74.3	12.8
4-grams	73.2	15.3
5-grams	70.8	18.2
Step 3 — Language		
Without	**77.5**	**11.8**
With	76.1	13.0
Step 4 — Lemmatization		
Without	**77.5**	**11.8**
With	76.8	11.8

(b) Linear SVM

Test	CV [%]	Bias [%]
Step 1 — Regularization		
$C = 1$	71.3	0.0
$C = 0.1$	74.0	0.0
$C = 0.05$	72.9	0.6
C=0.01	**74.1**	**5.0**
$C = 0.005$	73.3	8.2
$C = 0.001$	68.4	17.5
$C = 1e-4$	65.7	25.0
Step 2 — N-Grams		
2-grams	74.1	5.0
3-grams	**74.4**	**5.8**
Step 3 — Language		
Without	72.3	5.8
With	**72.2**	**8.1**

(c) Decision Tree

Test	CV [%]	Bias [%]
Step 1 — Criterion		
entropy	55.6	0.0
gini	**56.1**	**0.0**
Step 2 — Max Depth		
50	**57.2**	**9.6**
100	56.1	0
500	55.6	0
1000	57.3	0
Step 3 — Min Samples Split		
2	**57.2**	**9.6**
5	55.9	10.4
10	53.1	13.2

From these results, it is now possible to select the model that will be used for the Kaggle competition. When selecting the model, it is important to consider the CV accuracy as well as the bias. A low bias i.e. a train accuracy of almost 100%, means that the model is overfitted. Based on this the following two models were chosen:

- Bernoulli Naïve Bayes, 2000 features, 2-grams and no language identification
- SVM, 2000 features, 3-grams with language identification

The test accuracies obtained are presented in Table 2.

Table 2: Best Models' Test Accuracies

Model	Test Accuracy [%]
Naïve Bayes	65.1
SVM	71.1

5 Discussion and Conclusion

Based on the results, SVM is the best model for the subreddit classification. The model's parameter and dataset pre-processing are reported below:

- N-grams: 2
- Feature selection: 2000 selected based on their ANOVA F-value
- Language identification: Yes
- Regularization: $C = 0.01$

This finding is surprising since, based on the cross-validation accuracy, it was expected that the Naïve Bayes classifier would perform better. However, its test accuracy is more than 10% lower while the one for SVM remained similar.

Processing the posts was one of the more delicate steps of the project. Because the posts were in two different languages, the processing of the text was complicated. To try to account for the different languages, an additional language feature was added. It was observed that the majority of the comments from the London or Toronto subreddits were English, and the majority of comments from the Montreal and Paris subreddits were French. An additional feature was added to represent the language as English or French. Although this additional feature had some advantages, there were issues with identifying posts from the minority language. For example, if there was an English post in the Montreal subreddit it would be classified as London or Toronto, despite some obvious indicators that it was from Montreal.

It is also interesting to analyze the feature probabilities of the Naïve Bayes classifier. As part of the training, the probability of occurrence of a certain feature for a given class is estimated by $P(x_j = 1|y = k) = \theta_{j,k}$. The words associated with the highest values of $\theta_{j,k}$ for the four classes are reported below.

Table 3: Most probable features for a given class

London	**Montreal**	**Paris**	**Toronto**
like	people	paris	people
london	montreal	plus	like
people	would	ca	one
get	get	si	toronto
also	like	tout	would

It is possible to note that some words, like "people", are present in more than one class and thus, provide little information to the classification. A way to improve the accuracy of the Naïve Bayes classifier would be to weigh the words according to the information it provides about the class. This is the idea behind class-specific feature weighting. Class-specific weighting assigns a different weight to the feature depending on its relevance in the respective class, as opposed to the same weight across all classes [7]. This approach would benefit the classification of the project since there are certain words that have more relevance for certain classes. For instance, a word like "eiffel" that is present only a few times (5) in the training dataset would be given a high weight since it was only observed when the class was "Paris". By adding more weight to this word specifically for this class, better accuracy could be achieved. Ruan *et al.* [7] propose multiple feature weighting approaches that could be applied to text classification in this project.

Additionally, there are important characters such as $, £, and € that are useful for classification. In addition to adding class-specific deep feature weighting to words, weight could be added to these features as well.

Another method for adding weight is term frequency x inverse document frequency (TF-IDF). TF-IDF adds weight to words that are more important in the text, meaning the word appears many times in the document and is a relatively rare word [8]. Since there are multiple classes in the sample, it would be interesting to add an additional component to the TF-IDF method to account for the frequency in a class compared to the overall dataset.

To conclude, two models with similar cross-validation performance were found: one using linear SVM and one using Bernoulli Naïve Bayes. However, finding the pre-processing that gave the best results was quite a challenging task that required a lot of intuition and time. For future work, it would be interesting to see if implementing class-specific feature weighting could reduce the dependence of the performances on this step.

6 Statement of Contribution

Charles created the Naïve Bayes algorithm and the k-fold validation code. He also implemented the automated process that compares the performances of the Naïve Bayes algorithm with the other two machine learning. Ashley created the code for preprocessing the features and wrote up the report.

References

[1] A. Narges, "Ecse 551 - machine learning for engineers: Lecture 11 – regularization, decision trees," 2023.
[2] A. Narges, "Ecse 551 - machine learning for engineers: Lecture 8 – naive bayes," 2023.
[3] J. D. Rennie, L. Shih, J. Teevan, and D. R. Karger, "Tackling the poor assumptions of naive bayes text classifiers," in *Proceedings of the 20th international conference on machine learning (ICML-03)*, 2003, pp. 616–623.
[4] N. Shuyo, *Language detection library for java*, 2010. [Online]. Available: `http://code.google.com/p/language-detection/`.
[5] [Online]. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html#sklearn.feature_selection.f_classif`.

[6] S. Yue, P. Li, and P. Hao, "Svm classification: Its contents and challenges," *Applied Mathematics-A Journal of Chinese Universities*, vol. 18, pp. 332–342, 2003.

[7] S. Ruan, H. Li, C. Li, and K. Song, "Class-specific deep feature weighting for naïve bayes text classifiers," *IEEE Access*, vol. 8, pp. 20 151–20 159, 2020.

[8] A. Narges, "Ecse 551 - machine learning for engineers: Lecture 13 – decision trees (cont'd), feature construction," 2023.

7 Appendix

7.1 MP2.ipynb

MP2

November 19, 2023

1 ECSE-551 Mini Project 2

Authors: * Ashley Meagher (260822930) * Charles Sirois (261158513)

[250]:
```
# To specify where to load the data
in_colab = True
folder_path = 'drive/MyDrive/Colab Notebooks/ECSE 551_MP2'

%load_ext autoreload
%autoreload 2

# Our functions and classes
if in_colab:
  from google.colab import drive
  from google.colab import data_table
  drive.mount('/content/drive')

  data_table.enable_dataframe_formatter()  # For interactive df viz

  import sys
  sys.path.insert(0, folder_path)

# SK Learn models
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import LinearSVC

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
import itertools
import datetime

import nltk
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
```

```
# Install required packages
!pip install unidecode  # To remove accents
!pip install langid  # To identify text's language

# Import our classes and functions from the other files
from NaiveBayes import NaiveBayes
from cross_val_score import cross_val_score
from data_processing import Data, Format_data
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
Requirement already satisfied: unidecode in /usr/local/lib/python3.10/dist-
packages (1.3.7)

[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data…
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!

Requirement already satisfied: langid in /usr/local/lib/python3.10/dist-packages
(1.1.6)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(from langid) (1.23.5)
```

1.1 Data Analysis

1.1.1 Load the data

```
[251]: print(f"Loading data files... ", end='')
filenames = [folder_path + "/data/train_utf8.csv", folder_path + "/data/
 ↪test_utf8.csv"]
words_dataset = Data(train_file=filenames[0], test_file=filenames[1])
print(f'Done')
```

```
Loading data files… Done
```

1.1.2 Data properties

[252]:
```
print(f'Training dataset size: {words_dataset.train_data.shape[0]}')
print(f'Test dataset size: {words_dataset.test_data.shape[0]}')

classes, classes_count = np.unique(words_dataset.train_data['label'],␣
 ↪return_counts=True)
print(f'Classes: ')
for cls, cls_count in zip(classes, classes_count):
  print(f'\t-{cls}: {cls_count}')
```

```
Training dataset size: 719
Test dataset size: 279
Classes:
        -London: 180
        -Montreal: 180
        -Paris: 180
        -Toronto: 179
```

1.2 Models Performances

1.2.1 Functions

Functions to compute the cross-validation score of the different combinations of model hyperparameters and datasets

[253]:
```
def create_datasets(ds_options_dict):
  """
  To create a list with all the combinations of options in the dict
  """
  print(f"Processing input data...")
  keys, values = zip(*ds_options_dict.items())
  ds_options_list = [dict(zip(keys, v)) for v in itertools.product(*values)]

  ds_list = []
  for idx, each_ds in enumerate(ds_options_list):
      each_ds['dataset_name'] = f'DS {idx}'
      ds_list.append(Format_data(words_dataset, **each_ds))

  print(f'\nDone')
  return ds_list

def find_ds_from_name(ds_name, ds_list) -> Format_data:
    """
    To return the dataset with the corresponding name in the `ds_list`
    """
    ds = next((ds for ds in ds_list if ds.name == ds_name), None)

    if ds is None:
```

```python
        raise ValueError(f"Dataset {ds_name} not found in `ds_list`")

    return ds

def compute_models_cv_acc(model_dict, ds_list):
    """
    To compute the cv score for all the combinations of model_dict and ds_list
    """
    results_df = pd.DataFrame()

    # Cross-Validation
    n_fold = 5

    start_time = time.time()
    print(f"--------- Training all models ---------")
    for model_name, model_info in model_dict.items():
        model = model_info["model"]
        base_params = model_info["base_params"]
        cv_params = model_info["cv_params"]

        print(f"\nModel : {model_name}")
        model_start = time.time()
        for ds_idx, each_dataset in enumerate(ds_list):
            # Check if it already has been ran
            ds_start = time.time()
            dataset_name = each_dataset.name
            print(f"\tDataset [{ds_idx+1}/{len(ds_list)}]: {dataset_name}")

            X_train = each_dataset.X
            y_train = each_dataset.Y

            # Cross_validation
            cv_results = cross_val_score(
                model,
                X_train,
                y_train,
                cv=n_fold,
                base_params=base_params,
                cv_params=cv_params,
                results_df=results_df,
                ds_name=dataset_name,
            )

            if cv_results.empty:
                print(f'... Model already trained')
                continue
            # Print best combination
            best_row = cv_results.iloc[cv_results['Score'].idxmax()]
            compute_time = time.time() - ds_start
            print(
                f"\tBest CV Score : {np.round(best_row['Score']*100)}% (Acc: {np.
    ↪round(best_row['Acc']*100)}) "
                f"[{compute_time} sec]\n"
            )

            # Add information to series
            ds_params = each_dataset.get_params()

            for key, value in ds_params.items():
                if isinstance(value, tuple):
                    value = str(value)
                cv_results[key] = value

            # cv_results = pd.concat([cv_results, pd.(ds_params).T],␣
    ↪ignore_index=True)
            cv_results['Model name'] = model_name
            cv_results['Dataset'] = dataset_name
            cv_results['Compute time'] = compute_time

            results_df = pd.concat([results_df, cv_results], ignore_index=True,␣
    ↪axis=0)

        print(f'Model trained in {time.time() - model_start} sec')

    print(f"\nTraining completed ({time.time() - start_time} sec)\n")

    results_df['Score'] = (results_df['Score']*100).apply(np.round, decimals=2)
    results_df['Bias'] = ((1 - results_df['Acc'])*100).apply(np.round, decimals=2)

    results_df = results_df[
        [
            'Model name',
            'Score',
            'Bias',
            'Acc',
            'Dataset',
            'Params',
            'Compute time',
            'Model',
            'n_gram',
            'feat_type',
            'lemmatized',
            'lang',
            'standardized',
            'rm_accents',
            'feat_select',
            'n_feat',
        ]
    ]


    results_df = results_df.sort_values(by=['Score'], ascending=False)

    return results_df

def create_pred_ds(model_idx, results_df, ds_list, save_path):
    """ To create the prediction csv file for the model correponding to␣
 ↪model_idx
    The file is saved under save_path
    """
    my_model_info = results_df.loc[model_idx]
    print(f'Model chosen: ')
    print(my_model_info)

    print(f"Predicting test data using this model...")
    my_model = my_model_info['Model']
    ds = find_ds_from_name(my_model_info['Dataset'], ds_list)

    y_test = my_model.predict(ds.X_test)
    pred_df = pd.DataFrame(y_test, columns=['subreddit'])
    pred_df.index.name = 'id'

    pred_df.to_csv(save_path)
    print(f'Predictions saved to {save_path}')
```

1.2.2 Bernouilli Naive Bayes

Parameters evaluation

[254]:
```
nb_ds_options = {
    'max_feat': [None],
    'lang_id': [False, True],  # [False, True],
    'feature_type': ['Bin'],
    'n_gram': [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)],
    'lemmatize': [False],
    'feat_select': ['F_CL'],
    'n_feat_select': [500, 1000, 2000, 3000, 4000],
}
nb_ds_list = create_datasets(nb_ds_options)
```

```
Processing input data…
        Processing of: DS 49…
Done
```

[255]:
```
# Separate datasets to test lemmatization
nb2_ds_options = {
    'max_feat': [None],
    'lang_id': [False],  # [False, True],
    'feature_type': ['Bin'],
    'n_gram': [(1, 2)],
    'lemmatize': [True, False],
    'feat_select': ['F_CL'],
    'n_feat_select': [2000],
}
nb2_ds_list = create_datasets(nb2_ds_options)
```

```
Processing input data…
        Processing of: DS 1…
Done
```

[256]:
```
nb_model_dict = {}
nb_model_dict["My Bernouilli NB"] = {
    "model": NaiveBayes,
    'base_params': {'laplace_smoothing': True, 'verbose': False},
    'cv_params': None,
}
```

[257]:
```
nb_df = compute_models_cv_acc(nb_model_dict, nb_ds_list)
nb_results = nb_df[['Score', 'Bias', 'Params', 'n_gram', 'lang', 'n_feat']]
```

```
--------- Training all models ---------

Model : My Bernouilli NB
        Dataset [1/50]: DS 0
        Combination 1/1 Best CV Score : 71.0% (Acc: 77.0) [4.215232849121094
sec]

        Dataset [2/50]: DS 1
        Combination 1/1 Best CV Score : 72.0% (Acc: 82.0) [8.167191982269287
sec]

        Dataset [3/50]: DS 2
        Combination 1/1 Best CV Score : 75.0% (Acc: 88.0) [20.4975848197937 sec]

        Dataset [4/50]: DS 3
        Combination 1/1 Best CV Score : 68.0% (Acc: 87.0) [26.63149333000183
sec]

        Dataset [5/50]: DS 4
        Combination 1/1 Best CV Score : 62.0% (Acc: 83.0) [37.83934044837952
sec]

        Dataset [6/50]: DS 5
        Combination 1/1 Best CV Score : 70.0% (Acc: 77.0) [5.771634101867676
sec]

        Dataset [7/50]: DS 6
        Combination 1/1 Best CV Score : 71.0% (Acc: 83.0) [7.290061712265015
sec]

        Dataset [8/50]: DS 7
        Combination 1/1 Best CV Score : 78.0% (Acc: 88.0) [18.11064648628235
sec]

        Dataset [9/50]: DS 8
        Combination 1/1 Best CV Score : 74.0% (Acc: 91.0) [29.609700202941895
sec]

        Dataset [10/50]: DS 9
        Combination 1/1 Best CV Score : 68.0% (Acc: 88.0) [36.51623868942261
sec]

        Dataset [11/50]: DS 10
        Combination 1/1 Best CV Score : 65.0% (Acc: 71.0) [4.02064323425293 sec]

        Dataset [12/50]: DS 11
        Combination 1/1 Best CV Score : 70.0% (Acc: 81.0) [10.331190824508667
sec]

        Dataset [13/50]: DS 12
        Combination 1/1 Best CV Score : 75.0% (Acc: 87.0) [18.814451694488525
sec]

        Dataset [14/50]: DS 13
        Combination 1/1 Best CV Score : 73.0% (Acc: 90.0) [29.117358684539795
sec]

        Dataset [15/50]: DS 14
        Combination 1/1 Best CV Score : 69.0% (Acc: 89.0) [37.6209557056427 sec]

        Dataset [16/50]: DS 15
        Combination 1/1 Best CV Score : 60.0% (Acc: 66.0) [3.796255588531494
sec]

        Dataset [17/50]: DS 16
        Combination 1/1 Best CV Score : 68.0% (Acc: 76.0) [11.544331073760986
sec]
```

```
        Dataset [18/50]: DS 17
        Combination 1/1 Best CV Score : 72.0% (Acc: 85.0) [17.99778699874878
sec]

        Dataset [19/50]: DS 18
        Combination 1/1 Best CV Score : 70.0% (Acc: 89.0) [28.106534481048584
sec]

        Dataset [20/50]: DS 19
        Combination 1/1 Best CV Score : 69.0% (Acc: 89.0) [36.949981927871704
sec]

        Dataset [21/50]: DS 20
        Combination 1/1 Best CV Score : 57.0% (Acc: 62.0) [3.6437978744506836
sec]

        Dataset [22/50]: DS 21
        Combination 1/1 Best CV Score : 64.0% (Acc: 74.0) [9.618980169296265
sec]

        Dataset [23/50]: DS 22
        Combination 1/1 Best CV Score : 69.0% (Acc: 82.0) [18.49955105781555
sec]

        Dataset [24/50]: DS 23
        Combination 1/1 Best CV Score : 69.0% (Acc: 86.0) [26.71236515045166
sec]

        Dataset [25/50]: DS 24
        Combination 1/1 Best CV Score : 68.0% (Acc: 88.0) [36.23162865638733
sec]

        Dataset [26/50]: DS 25
        Combination 1/1 Best CV Score : 69.0% (Acc: 75.0) [6.965734243392944
sec]

        Dataset [27/50]: DS 26
        Combination 1/1 Best CV Score : 70.0% (Acc: 81.0) [7.405640363693237
sec]

        Dataset [28/50]: DS 27
        Combination 1/1 Best CV Score : 74.0% (Acc: 87.0) [17.702255725860596
sec]

        Dataset [29/50]: DS 28
        Combination 1/1 Best CV Score : 67.0% (Acc: 85.0) [28.42513680458069
sec]
```

```
        Dataset [30/50]: DS 29
        Combination 1/1 Best CV Score : 63.0% (Acc: 82.0) [36.266666412353516
sec]

        Dataset [31/50]: DS 30
        Combination 1/1 Best CV Score : 69.0% (Acc: 75.0) [3.7581331729888916
sec]

        Dataset [32/50]: DS 31
        Combination 1/1 Best CV Score : 73.0% (Acc: 83.0) [10.553314924240112
sec]

        Dataset [33/50]: DS 32
        Combination 1/1 Best CV Score : 75.0% (Acc: 87.0) [17.867765426635742
sec]

        Dataset [34/50]: DS 33
        Combination 1/1 Best CV Score : 74.0% (Acc: 88.0) [26.27276086807251
sec]

        Dataset [35/50]: DS 34
        Combination 1/1 Best CV Score : 67.0% (Acc: 87.0) [37.33134150505066
sec]

        Dataset [36/50]: DS 35
        Combination 1/1 Best CV Score : 62.0% (Acc: 69.0) [4.442660331726074
sec]

        Dataset [37/50]: DS 36
        Combination 1/1 Best CV Score : 70.0% (Acc: 79.0) [7.428954124450684
sec]

        Dataset [38/50]: DS 37
        Combination 1/1 Best CV Score : 75.0% (Acc: 86.0) [18.159637451171875
sec]

        Dataset [39/50]: DS 38
        Combination 1/1 Best CV Score : 72.0% (Acc: 88.0) [28.107608318328857
sec]

        Dataset [40/50]: DS 39
        Combination 1/1 Best CV Score : 69.0% (Acc: 87.0) [35.64508605003357
sec]

        Dataset [41/50]: DS 40
        Combination 1/1 Best CV Score : 59.0% (Acc: 64.0) [4.095541715621948
sec]
```

```
        Dataset [42/50]: DS 41
        Combination 1/1 Best CV Score : 66.0% (Acc: 75.0) [10.126330375671387
sec]

        Dataset [43/50]: DS 42
        Combination 1/1 Best CV Score : 72.0% (Acc: 84.0) [17.864989519119263
sec]

        Dataset [44/50]: DS 43
        Combination 1/1 Best CV Score : 70.0% (Acc: 87.0) [27.56903648376465
sec]

        Dataset [45/50]: DS 44
        Combination 1/1 Best CV Score : 69.0% (Acc: 88.0) [37.22462606430054
sec]

        Dataset [46/50]: DS 45
        Combination 1/1 Best CV Score : 56.0% (Acc: 60.0) [3.685528516769409
sec]

        Dataset [47/50]: DS 46
        Combination 1/1 Best CV Score : 64.0% (Acc: 72.0) [9.841209650039673
sec]

        Dataset [48/50]: DS 47
        Combination 1/1 Best CV Score : 69.0% (Acc: 81.0) [18.502774715423584
sec]

        Dataset [49/50]: DS 48
        Combination 1/1 Best CV Score : 67.0% (Acc: 84.0) [25.65246319770813
sec]

        Dataset [50/50]: DS 49
        Combination 1/1 Best CV Score : 69.0% (Acc: 87.0) [35.62777662277222
sec]

Model trained in 964.5682625770569 sec

Training completed (964.5693309307098 sec)
```

[258]:
```
nb2_df = compute_models_cv_acc(nb_model_dict, nb2_ds_list)
nb2_results = nb2_df[['Score', 'Bias', 'Params', 'n_gram', 'lang', 'n_feat',␣
 ↪'lemmatized']]
```

```
--------- Training all models ---------
```

```
Model : My Bernouilli NB
        Dataset [1/2]: DS 0
        Combination 1/1 Best CV Score : 75.0% (Acc: 88.0) [18.69008207321167
sec]

        Dataset [2/2]: DS 1
        Combination 1/1 Best CV Score : 79.0% (Acc: 88.0) [19.127805709838867
sec]

Model trained in 37.849053144454956 sec

Training completed (37.85080671310425 sec)
```

Step 1 - Effect of feature selection

```
[259]: nb_results_step1 = nb_results[(nb_results['n_gram'] == '(1, 1)') &␣
        ↪(nb_results['lang'] == False)]
       nb_results_step1.sort_values(by=['n_feat'], ascending=True)
```

```
[259]:    Score   Bias Params  n_gram   lang  n_feat
       0  71.36  22.81     {}  (1, 1)  False     500
       1  71.63  18.36     {}  (1, 1)  False    1000
       2  75.11  11.54     {}  (1, 1)  False    2000
       3  67.59  13.21     {}  (1, 1)  False    3000
       4  61.89  16.97     {}  (1, 1)  False    4000
```

Step 2 - N-grams

```
[260]: nb_results_step2 = nb_results[(nb_results['n_feat'] == 2000) &␣
        ↪(nb_results['lang'] == False)]
       nb_results_step2.sort_values(by=['n_gram'], ascending=True)
```

```
[260]:     Score   Bias Params  n_gram   lang  n_feat
       2   75.11  11.54     {}  (1, 1)  False    2000
       7   78.17  11.82     {}  (1, 2)  False    2000
       12  74.83  12.80     {}  (1, 3)  False    2000
       17  72.05  15.30     {}  (1, 4)  False    2000
       22  69.12  18.22     {}  (1, 5)  False    2000
```

Step 3 - Language Identification

```
[261]: nb_results_step3 = nb_results[(nb_results['n_feat'] == 2000) &␣
        ↪(nb_results['n_gram'] == '(1, 2)')]
       nb_results_step3
```

```
[261]:     Score   Bias Params  n_gram   lang  n_feat
       7   78.17  11.82     {}  (1, 2)  False    2000
       32  75.25  12.93     {}  (1, 2)   True    2000
```

Step 4 - Effect of lemmatization

[262]:
```
nb_results_step4 = nb2_results
nb_results_step4
```

[262]:
```
   Score   Bias Params  n_gram   lang  n_feat  lemmatized
1  78.73  11.82     {}  (1, 2)  False    2000       False
0  75.11  11.82     {}  (1, 2)  False    2000        True
```

1.2.3 SVC

[263]:
```
svc_ds_options = {
    'max_feat': [None],
    'lang_id': [False, True],  # [False, True],
    'feature_type': ['Count'],
    'n_gram': [(1, 2), (1, 3)],
    'lemmatize': [False],
    'feat_select': ['F_CL'],
    'n_feat_select': [2000],
}
svc_ds_list = create_datasets(svc_ds_options)

svc_model_dict = {}
svc_model_dict["SVC"] = {
    "model": LinearSVC,
    "base_params": {"random_state": 0},
    "cv_params": {"C": [0.0001, 0.001, 0.005, 0.01, 0.05, 0.1, 1]},
}
```

```
Processing input data…
        Processing of: DS 3…
Done
```

[264]:
```
svc_df = compute_models_cv_acc(svc_model_dict, svc_ds_list)
svc_results = svc_df[['Score', 'Bias', 'Params', 'n_gram', 'lang', 'n_feat']]
```

```
--------- Training all models ---------

Model : SVC
        Dataset [1/4]: DS 0
        Combination 7/7 Best CV Score : 74.0% (Acc: 95.0) [0.5722982883453369
sec]

        Dataset [2/4]: DS 1
        Combination 7/7 Best CV Score : 73.0% (Acc: 99.0) [0.7092363834381104
sec]

        Dataset [3/4]: DS 2
        Combination 7/7 Best CV Score : 74.0% (Acc: 99.0) [0.5993454456329346
sec]

        Dataset [4/4]: DS 3
        Combination 7/7 Best CV Score : 74.0% (Acc: 99.0) [1.3745067119598389
sec]

Model trained in 3.2752864360809326 sec

Training completed (3.2758843898773193 sec)
```

[265]:
```
svc_results
```

[265]:
```
    Score   Bias          Params  n_gram   lang  n_feat
3   74.13   5.01    {'C': 0.01}  (1, 2)  False    2000
4   73.99   0.56    {'C': 0.05}  (1, 2)  False    2000
18  73.85   0.70    {'C': 0.05}  (1, 2)   True    2000
25  73.85   0.83    {'C': 0.05}  (1, 3)   True    2000
2   73.57   8.21   {'C': 0.005}  (1, 2)  False    2000
5   73.44   0.00     {'C': 0.1}  (1, 2)  False    2000
11  73.16   0.70    {'C': 0.05}  (1, 3)  False    2000
19  72.60   0.00     {'C': 0.1}  (1, 2)   True    2000
12  72.47   0.14     {'C': 0.1}  (1, 3)  False    2000
17  72.47   7.93    {'C': 0.01}  (1, 2)   True    2000
9   72.47   8.21   {'C': 0.005}  (1, 3)  False    2000
10  72.32   5.84    {'C': 0.01}  (1, 3)  False    2000
26  72.18   0.14     {'C': 0.1}  (1, 3)   True    2000
24  71.90   8.07    {'C': 0.01}  (1, 3)   True    2000
16  70.93  12.52   {'C': 0.005}  (1, 2)   True    2000
27  70.52   0.00       {'C': 1}  (1, 3)   True    2000
20  70.51   0.00       {'C': 1}  (1, 2)   True    2000
13  69.54   0.00       {'C': 1}  (1, 3)  False    2000
23  69.12  12.38   {'C': 0.005}  (1, 3)   True    2000
6   68.99   0.00       {'C': 1}  (1, 2)  False    2000
8   68.01  17.39   {'C': 0.001}  (1, 3)  False    2000
15  67.31  22.67   {'C': 0.001}  (1, 2)   True    2000
1   66.20  17.52   {'C': 0.001}  (1, 2)  False    2000
7   65.49  25.31  {'C': 0.0001}  (1, 3)  False    2000
22  65.09  22.67   {'C': 0.001}  (1, 3)   True    2000
0   64.25  25.03  {'C': 0.0001}  (1, 2)  False    2000
14  59.66  31.29  {'C': 0.0001}  (1, 2)   True    2000
21  59.39  31.57  {'C': 0.0001}  (1, 3)   True    2000
```

Step 1 - Regularization

[266]:
```
svc_results_step1 = svc_results[(svc_results['n_feat'] == 2000) &␣
 ↪(svc_results['n_gram'] == '(1, 2)') & (svc_results['lang'] == False)]
svc_results_step1
```

```
[266]:    Score   Bias          Params  n_gram   lang  n_feat
       3  74.13   5.01    {'C': 0.01}  (1, 2)  False    2000
       4  73.99   0.56    {'C': 0.05}  (1, 2)  False    2000
       2  73.57   8.21   {'C': 0.005}  (1, 2)  False    2000
       5  73.44   0.00     {'C': 0.1}  (1, 2)  False    2000
       6  68.99   0.00       {'C': 1}  (1, 2)  False    2000
       1  66.20  17.52   {'C': 0.001}  (1, 2)  False    2000
       0  64.25  25.03  {'C': 0.0001}  (1, 2)  False    2000
```

Step 2 - N-Grams

```
[267]: svc_results_step2 = svc_results[(svc_results['n_feat'] == 2000) &␣
        ↪(svc_results['Params'] == {'C': 0.01}) & (svc_results['lang'] == False)]
       svc_results_step2
```

```
[267]:     Score  Bias       Params  n_gram   lang  n_feat
       3   74.13  5.01  {'C': 0.01}  (1, 2)  False    2000
       10  72.32  5.84  {'C': 0.01}  (1, 3)  False    2000
```

Step 3 - Language

```
[268]: svc_results_step3 = svc_results[(svc_results['n_feat'] == 2000) &␣
        ↪(svc_results['Params'] == {'C': 0.01}) & (svc_results['n_gram'] == '(1, 3)')]
       svc_results_step3
```

```
[268]:     Score  Bias       Params  n_gram   lang  n_feat
       10  72.32  5.84  {'C': 0.01}  (1, 3)  False    2000
       24  71.90  8.07  {'C': 0.01}  (1, 3)   True    2000
```

1.2.4 Decision Tree

```
[269]: dt_ds_options = {
           'max_feat': [None],
           'lang_id': [False],   # [False, True],
           'feature_type': ['Count'],
           'n_gram': [(1, 2)],
           'lemmatize': [False],
           'feat_select': ['F_CL'],
           'n_feat_select': [2000],
       }
       dt_ds_list = create_datasets(dt_ds_options)

       dt_model_dict = {}
       dt_model_dict["DT"] = {
           "model": DecisionTreeClassifier,
           "base_params": {"random_state": 0},
           "cv_params": {"criterion": ['gini', 'entropy'],
                         "max_depth": [50, 100, 500, 1000],
                    "min_samples_split": [2, 5, 10]},
}
```

```
Processing input data…
        Processing of: DS 0…
Done
```

[270]:
```
dt_df = compute_models_cv_acc(dt_model_dict, dt_ds_list)

dt_df['criterion'] = None
dt_df['max_depth'] = None
dt_df['min_samples_split'] = None

for idx, each_row in dt_df.iterrows():
  for key, val in each_row['Params'].items():
    dt_df.at[idx, key] = val

dt_results = dt_df[['Score', 'Bias', 'criterion','max_depth',␣
 ↪'min_samples_split', 'n_gram', 'lang', 'n_feat']]
```

```
--------- Training all models ---------

Model : DT
        Dataset [1/1]: DS 0
        Combination 24/24       Best CV Score : 59.0% (Acc: 100.0)
[8.762712717056274 sec]

Model trained in 8.768966913223267 sec

Training completed (8.770277261734009 sec)
```

Step 1 - Criterion

[271]:
```
dt_results_step1 = dt_results[(dt_results['n_feat'] == 2000) &␣
 ↪(dt_results['max_depth'] == 100) & (dt_results['min_samples_split'] == 2)]
dt_results_step1
```

[271]:
```
    Score  Bias criterion max_depth min_samples_split  n_gram   lang  n_feat
3   58.28   0.0      gini       100                 2  (1, 2)  False    2000
15  54.39   0.0   entropy       100                 2  (1, 2)  False    2000
```

Step 2 - Max Depth

[272]:
```
dt_results_step2 = dt_results[(dt_results['n_feat'] == 2000) &␣
 ↪(dt_results['criterion'] == 'gini') & (dt_results['min_samples_split'] == 2)]
dt_results_step2
```

[272]:
```
   Score  Bias criterion max_depth min_samples_split  n_gram   lang  n_feat
9  58.55   0.0      gini      1000                 2  (1, 2)  False    2000
3  58.28   0.0      gini       100                 2  (1, 2)  False    2000
6  57.17   0.0      gini       500                 2  (1, 2)  False    2000
0  55.91   9.6      gini        50                 2  (1, 2)  False    2000
```

Step 3 - Min Samples Split

[273]:
```
dt_results_step3 = dt_results[(dt_results['n_feat'] == 2000) &␣
 ↪(dt_results['criterion'] == 'gini') & (dt_results['max_depth'] == 50)]
dt_results_step3
```

[273]:
```
   Score   Bias criterion max_depth min_samples_split  n_gram   lang  n_feat
1  56.47  10.43      gini        50                 5  (1, 2)  False    2000
0  55.91   9.60      gini        50                 2  (1, 2)  False    2000
2  51.87  13.21      gini        50                10  (1, 2)  False    2000
```

1.2.5 Final Model

[274]:
```
best_nb_idx = 7
best_svc_idx = 24
```

[275]:
```
create_pred_ds(best_nb_idx, nb_df, nb_ds_list, folder_path + '/
 ↪NB_Final_prediction.csv')
```

```
Model chosen:
Model name                                         My Bernouilli NB
Score                                                         78.17
Bias                                                          11.82
Acc                                                         0.88178
Dataset                                                        DS 7
Params                                                           {}
Compute time                                              18.110646
Model          <NaiveBayes.NaiveBayes object at 0x78dce2ed2d40>
n_gram                                                       (1, 2)
feat_type                                                       Bin
lemmatized                                                    False
lang                                                          False
standardized                                                  False
rm_accents                                                     True
feat_select                                                    F_CL
n_feat                                                         2000
Name: 7, dtype: object
Predicting test data using this model…
Predictions saved to drive/MyDrive/Colab Notebooks/ECSE
551_MP2/NB_Final_prediction.csv
```

```
[276]: create_pred_ds(best_svc_idx, svc_df, svc_ds_list, folder_path + '/
       ↪SVC_Final_prediction.csv')
```

```
Model chosen:
Model name                                        SVC
Score                                            71.9
Bias                                             8.07
Acc                                          0.919332
Dataset                                          DS 3
Params                                    {'C': 0.01}
Compute time                                 1.374507
Model            LinearSVC(C=0.01, random_state=0)
n_gram                                         (1, 3)
feat_type                                       Count
lemmatized                                      False
lang                                             True
standardized                                    False
rm_accents                                       True
feat_select                                      F_CL
n_feat                                           2000
Name: 24, dtype: object
Predicting test data using this model…
Predictions saved to drive/MyDrive/Colab Notebooks/ECSE
551_MP2/SVC_Final_prediction.csv
```

1.3 Features Analysis

Check the words that are most probably observed for a given class

```
[277]: nb_info = nb_df.loc[best_nb_idx]
       nb_model = nb_info['Model']
       nb_model_ds = find_ds_from_name(nb_info['Dataset'], nb_ds_list)
       features_name = nb_model_ds.features_name

       n_best_features = 10

       df_dict = {}
       for k, class_label in enumerate(nb_model._classes):
           feats_score = nb_model._thetas[k, 1::]
           names_scores = list(zip(features_name, feats_score))
           feat_scores_df = pd.DataFrame(data=names_scores, columns=['Feat_names',␣
       ↪'Score'])
           feat_scores_df = feat_scores_df.sort_values(by=['Score'], ascending=False).
       ↪reset_index(drop=True)
           df_dict[class_label] = feat_scores_df

       combined_df = pd.concat(df_dict, axis=1)
       # print(combined_df.head(n_feats).to_string())
```

```
combined_df.head(n_best_features)
```

```
[277]:       London              Montreal                Paris               Toronto  \
        Feat_names     Score Feat_names     Score Feat_names     Score Feat_names
     0        like  0.296703     people  0.225275      paris  0.318681     people
     1      london  0.280220   montreal  0.186813       plus  0.269231       like
     2      people  0.263736      would  0.175824         ca  0.263736        one
     3         one  0.252747        get  0.175824         si  0.203297    toronto
     4         get  0.225275       like  0.159341       tout  0.181319      would
     5        also  0.192308        one  0.153846       etre  0.164835       city
     6      really  0.159341         ca  0.142857      faire  0.142857       also
     7       would  0.153846       good  0.126374      quand  0.142857        get
     8        know  0.153846         go  0.120879      comme  0.131868       time
     9         see  0.142857       time  0.115385       fait  0.126374        new


           Score
     0  0.248619
     1  0.243094
     2  0.204420
     3  0.198895
     4  0.187845
     5  0.171271
     6  0.165746
     7  0.160221
     8  0.160221
     9  0.149171
```

7.2 NaiveBayes.py

```python
from typing import Literal
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import pandas as pd
import time


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def cost_function(X, y, w):
    n = len(y)
    h = sigmoid(np.dot(X, w))
    J = (-1 / n) * (np.dot(y.T, np.log(h)) + np.dot((1 - y).T, np.log(1 - h)))
    return J


class NaiveBayes:
    """Naive Bayes class.

    If k is different than 0, will find the best alpha, tol and reg_cst
    using k-fold cross-validation.

    Attributes:
        thetas (ndarray): ndarray of shape (k, (1 + m))
          Theta values for classification
        class: Output class
        n_features (int): Dimension of features (m). 0 if model not fitted.
        n_iter (int): Number of iterations needed for fitting. 0 if not fitted.

       X (ndarray): Training inputs (nxm)
        y (ndarray): Training output (nx1)
    """

    def __init__(
        self,
        laplace_smoothing: bool = True,
        verbose=False,  # To print execution info
    ) -> None:
        self._n_features = 0

        self._classes = None
        self._n_class = 0
        self._class_count = None
        self._n_samples = 0

        self.X = None  # X dataset for training
        self.y = None  # Y dataset for training

        self.laplace_smoothing = laplace_smoothing

        self._log_class_prior = None
        self._feat_log_proba = None
        self._feat_log_proba = None

        self.results = None  # Results dataframe
        self._comp_time = None

        self._verbose = verbose

    def fit(self, X, y):
        """
        Fit the model according to the given training data.

        Parameters:
            X (ndarray) : shape (n_samples, n_features)
                Training vector.

            y (ndarray) : shape (n_samples,)
                Expected output vector

            w (ndarray, optional): shape (n_features, )
            T   o give an initial guess

        Returns:
            self
                Model with weights fitted to training dataset
        """
        self.X = X
        self.y = y

        # Check dataset sizes
        self._n_samples, self._n_features = X.shape

        if X.shape[0] != y.shape[0]:
            raise ValueError(f"Mismatch between the size of the input ({X.shape[0]})
    and outputs ({y.shape[0]})")

        # Number of class
        self._classes, self._class_count = np.unique(self.y, return_counts=True)
        self._n_class = len(self._classes)

        if self._verbose:
            print(f"Fitting Naive Bayes model for the dataset")
            print(f"\t# Features: {self._n_features}, classes: {self._classes}, #
    Samples: {self._n_samples}")

        self._train_model()

        return self

    def predict(self, X):
        """To predict the output of samples.

        For each class, computes:
            $$  \delta_k = \log{[\theta_k\sum_{j=1}^m \theta_{j, k}^{x_j} (1 - \theta_
    {j, k})^{1-x_j} ]}$$

        Classify the output as:
            $$ \text{Output} = \argmax_k \delta_k(x) $$

        Args:
            X (ndarray): Inputs sample to be predicted. Size (nxm)

        Raises:
            ValueError: If model is not trained

        Returns:
            ndarray: Predicted output of each data point (nx1)
        """
        if self._thetas is None:
            raise ValueError(f"Model is not trained.")

        log_class_prior = np.log(self._thetas[:, 0])  # log( P(Y=k) )  shape: 1xk
        feat_log_proba = np.log(self._thetas[:, 1::])  # log( P(x_j=1 | Y=k) )  shape:
     kxm
```

```python
        feat_log_neg_proba = np.log(1 - self._thetas[:, 1::])  # log( 1 - P(x_j=1 | Y=k) )  shape: kxm

        n_samples = X.shape[0]

        self._joint_log_likelihood = np.zeros((n_samples, self._n_class))

        self._joint_log_likelihood = X @ (feat_log_proba - feat_log_neg_proba).T + feat_log_neg_proba.sum(axis=1)

        # Add class log priors
        self._joint_log_likelihood += log_class_prior

        # Predictions
        predictions_idx = np.argmax(self._joint_log_likelihood, axis=1)
        predictions = self._classes[predictions_idx]

        return predictions

    def score(self, X, y):
        """To compute the accuracy of the model

        Args:
            X (ndarray): Test samples
            y (ndarray): True class of X

        Returns:
             float: Accuracy of the model over test samples
        """
        y_pred = self.predict(X)

        accuracy = (y == y_pred).mean()

        return accuracy

    def _train_model(self):
        """Compute the theta needed to estimate the probabilities

        For each class:
            $$ \theta_k = P(Y=k) = (# samples where Y=k) / (# samples) $$
            $$ \theta_{j, k} = P(x_j=1 | Y=k) = (# samples where x_j=1 and Y=k) / (# samples where Y=k)  $$

        Stores the values in a np.array of shape k x (1 + m)
            -> \theta_{k} = thetas[k, 0], k=0, ... n_class  (prior of class k)
            -> \theta_{j, k} = thetas[k, j], j=1, ..., m
        """
        self._thetas = np.zeros([self._n_class, self._n_features + 1])

        for k, class_label in enumerate(self._classes):
            n_yk = self._class_count[k]  # n samples where Y=k

            X_k = self.X[self.y == class_label, :]

            theta_k = n_yk / self._n_samples  # P(Y=k), prior for class k
            self._thetas[k, 0] = theta_k

            for j in range(self._n_features):
                samples_j_k = X_k[:, j] == 1  # Array with True where X_j is 1

                n_xj_yk = samples_j_k.sum()  # n samples where Y=k and x=x_j

                if self.laplace_smoothing:
                    theta_j_k = (n_xj_yk + 1) / (n_yk + 2)
                else:
                    theta_j_k = n_xj_yk / n_yk

                self._thetas[k, j + 1] = theta_j_k  # \theta_{j, k}
```

7.3 data_processing.py

```
import pandas as pd
import numpy as np
import scipy.sparse as sp
import matplotlib.pyplot as plt
from sklearn.feature_extraction import text
from nltk import word_tokenize
import pickle
from functools import partial
from typing import Literal
import unidecode

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

import string
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

import nltk
from nltk.corpus import wordnet
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer

import langid
from langid.langid import LanguageIdentifier, model

langid.set_languages(['en', 'fr'])
lang_identifier = LanguageIdentifier.from_modelstring(model, norm_probs=True)

MY_STOP_WORDS = ['im', 'https', 'http', 'www', '1', 're', 'qu', 'x200b']


def get_wordnet_pos(word):
    """Map POS tag to first character lemmatize() accepts"""
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ, "N": wordnet.NOUN, "V": wordnet.VERB, "R": wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)


class LemmaTokenizer:
    def __init__(self):
        self.wnl = WordNetLemmatizer()

    def __call__(self, doc):
        return [self.wnl.lemmatize(t, pos=get_wordnet_pos(t)) for t in word_tokenize(doc) if t.isalpha()]


def MyTokenizer(text):
    """
    To keep $ and pound signs
    """
    text = text.split()

    important_symbols = ['$', ' ', ' ']
    for symb in important_symbols:
        if any(symb in string for string in text):
            text.append(symb)

    return text


class Data:
    def __init__(self, train_file, test_file):
        # Download the csv data
        self.train_file: str = train_file
        self.test_file: str = test_file

        self.train_data: pd.DataFrame
        self.test_data: pd.DataFrame
        self.readData()
        self._detect_lang()

        # Extract the subsets
        self.data_list: list = self.train_data['body'].to_list()  # List of all samples

        self.labels = self.train_data['label'].to_numpy()  # Numpy array with label of each sample

    # Read the data
    def readData(self):
        self.train_data = pd.read_csv(
            self.train_file, header=None, encoding='utf-8', skiprows=[0], names=['body', 'label']
        )
        self.train_data = self.train_data.sample(frac=1, random_state=0)
        self.test_data = pd.read_csv(self.test_file, header=None, encoding='utf-8', skiprows=[0], names=['id', 'body'])

    def _detect_lang(self):
        """
        To find the language (fr or en) of each post
        """
        self.train_data['lang'] = self.train_data['body'].apply(lambda x: lang_identifier.classify(x)[0])
        self.test_data['lang'] = self.test_data['body'].apply(lambda x: lang_identifier.classify(x)[0])


class Format_data:
    def __init__(
        self,
        words_dataset: Data,  # Loaded data
        dataset_name: str = 'NoName',
        # Text processing options
        max_feat: int | None = None,  #  Max number of tokens
        feature_type: Literal['Bin', 'Count', 'TF'] = 'Bin',
        n_gram: tuple = (1, 1),
        lemmatize: bool = False,
        lang_id: bool = False,  # If true, add a feature for the language (0: en, 1:fr)
        rm_accents: bool = True,  # To remove accents
        standardize_data: bool = False,  # To remove mean and std of all data
        min_df: int = 1,  # Ignore terms w/ frequency lower than that
        # Feature selection options
        feat_select: Literal['PCA', 'MI', 'F_CL'] | None = None,
        n_feat_select: int = 1,  # Number of features to keep
        weight_samples: bool = False,  # To compute the features weights
        punc_replace: str = ' ',
    ):
        self.name: str = dataset_name
        print(f"\r\tProcessing of: {self.name}... ", end='')

        # Attributes
        self.words_dataset: Data = words_dataset

        # Text processing
        self._max_feat = max_feat
        self._n_gram = n_gram

        self._feature_type = feature_type
        if feature_type == 'Bin':
            self._binary_features = True
            self._use_tf_idf = False

        elif feature_type == 'Count':
            self._binary_features = False
            self._use_tf_idf = False

        elif feature_type == 'TF':
            self._binary_features = False
            self._use_tf_idf = True

        self._lemmatize = lemmatize
        self._lang_id = lang_id
        self._standardize_data = standardize_data
        self._rm_accents = rm_accents
        self._min_df = min_df
        self._punc_rep = punc_replace

        # Feature selection
        self._feat_select_opt = feat_select
        self._n_feat_select = n_feat_select

        # Train labels
        self.Y = words_dataset.labels
        self.X_test = None

        self.stop_words = self._get_stop_words()

        # Pre-process
        (
            self.train_text,
            self.test_text,
        ) = self._pre_process_text()  # Get list of posts, lowered and w/o
    punctuations

        # Tokenize
        self.X, self.X_test, self._vectorizer = self._vectorize_text()  # _vectorizer:
     To transform text to a vector

        self.features_name = self._vectorizer.get_feature_names_out()  # Corresponding
     features of _vectorizer

        self._add_lang()  # Add language as a feature

        self._scaler = self._normalize_data()

        # Feat. Selection
        self.pca_selector = None  # PCA transformer
        self.mi_selector = None  # MI feature selection
        self._feat_selector = self._feature_selection()

    def _vectorize_text(self):
        """
        Create a dictionary of all words.

        To get the CountVectorizer for the training dataset.

        Returns:
            _vectorizer and vectorized dataset

        """
        # Set tokenizer
        if self._lemmatize:
            tokenizer = LemmaTokenizer()

        else:
            tokenizer = MyTokenizer

        strip_accents = 'unicode' if self._rm_accents else None

        if not self._use_tf_idf:
            vectorizer = CountVectorizer(
                stop_words=self.stop_words,
                max_features=self._max_feat,
                ngram_range=self._n_gram,
                binary=self._binary_features,
                tokenizer=tokenizer,
                token_pattern=None,
                strip_accents=strip_accents,
                min_df=self._min_df,
            )

        else:
            vectorizer = TfidfVectorizer(
                stop_words=self.stop_words,
                max_features=self._max_feat,
                ngram_range=self._n_gram,
                binary=False,
                tokenizer=tokenizer,
                token_pattern=None,
                strip_accents=strip_accents,
                min_df=self._min_df,
            )

        # Learn the vocabulary dictionary and return document term matrix
        X = vectorizer.fit_transform(self.train_text)

        # Transform test data
        X_test = vectorizer.transform(self.test_text)

        return X, X_test, vectorizer

    def _pre_process_text(self):
        """
        Pre-process the texts:
            - Lowers everything
            - Remove punctuations

        Returns:
            [str]: List with all the post preprocessed

        """
        train_df = self.words_dataset.train_data.copy(deep=True)
        test_df = self.words_dataset.test_data.copy(deep=True)

        # Lower
        train_df['body'] = train_df['body'].str.lower()
        test_df['body'] = test_df['body'].str.lower()

        # Punctuation
        punc_list = string.punctuation.replace('$', '')
        punc_list += '              '

        train_df['body'] = train_df['body'].str.replace('[{}]'.format(punc_list), self._punc_rep, regex=True)
        train_df['body'] = train_df['body'].str.replace(r'[\n\\]', '', regex=True)

        test_df['body'] = test_df['body'].str.replace('[{}]'.format(punc_list), self._punc_rep, regex=True)
        test_df['body'] = test_df['body'].str.replace(r'[\n\\]', '', regex=True)

        return train_df['body'].to_list(), test_df['body'].to_list()

    # Specify stopwords
    def _get_stop_words(self):
        my_stop_words = stopwords.words('english') + stopwords.words('french')

        my_stop_words += MY_STOP_WORDS

        if self._rm_accents:
            my_stop_words = [unidecode.unidecode(word) for word in my_stop_words]

        # Lemmatize stop words
        if self._lemmatize:
            wnl = WordNetLemmatizer()
            my_stop_words = [wnl.lemmatize(t, pos=get_wordnet_pos(t)) for t in my_stop_words if t.isalpha()]
            my_stop_words = list(set(my_stop_words))

        # print(my_stop_words)
        return my_stop_words

    def _feature_selection(self):
        """
        To perform feature selection analysis
        """
        feat_selector = None
        # PCA
        if self._feat_select_opt == 'PCA':
            pca_selector = PCA(n_components=self._n_feat_select)

            if isinstance(self.X, sp.csr_matrix):
                self.X = self.X.toarray()

            self.X = pca_selector.fit_transform(self.X)
            self.X_test = pca_selector.transform(self.X_test)

            plot = True
            if plot:
                sing_values = pca_selector.singular_values_

                # Plot the singular values
                plt.plot(np.arange(1, len(sing_values) + 1), sing_values, marker='o')
                plt.title(f'Singular Values - {self.name}')
                plt.xlabel('Principal Components')
                plt.ylabel('Singular Values')
                plt.axvline(x=self._n_feat_select, color='red', linestyle='--', ymin=0, ymax=1, linewidth=2)
                plt.grid(True)
                plt.show(block=False)

            feat_selector = pca_selector
```

```python

        elif self._feat_select_opt == 'MI':
            if self._use_tf_idf:
                discrete_feat = [self.X.shape[1] - 1]
                X = self.X.toarray()
            else:
                discrete_feat = True
                X = self.X

            # MI_info = mutual_info_classif(X=self.X.toarray(), Y=self.Y, discrete_features=discrete_features, random_state=0)
            my_score = partial(mutual_info_classif, random_state=0, discrete_features=discrete_feat)
            mi_selector = SelectKBest(my_score, k=self._n_feat_select)
            self.X = mi_selector.fit_transform(X, self.Y)
            self.X_test = mi_selector.transform(self.X_test)

            selected_feats = self.features_name[mi_selector.get_support()]
            feat_scores = mi_selector.scores_[mi_selector.get_support()]
            names_scores = list(zip(selected_feats, feat_scores))
            feat_scores = pd.DataFrame(data=names_scores, columns=['Feat_names', 'Score'])
            self._feat_scores = feat_scores.sort_values(['Score', 'Feat_names'], ascending=[False, True])

            self.features_name = mi_selector.get_feature_names_out(self.features_name)

            feat_selector = mi_selector

        elif self._feat_select_opt == 'F_CL':
            feat_selector = SelectKBest(f_classif, k=self._n_feat_select)
            X_trans = feat_selector.fit_transform(self.X, self.Y)

            X_test_trans = feat_selector.transform(self.X_test)

            selected_feats = self.features_name[feat_selector.get_support()]
            feat_scores = feat_selector.scores_[feat_selector.get_support()]
            names_scores = list(zip(selected_feats, feat_scores))
            feat_scores = pd.DataFrame(data=names_scores, columns=['Feat_names', 'Score'])
            self._feat_scores = feat_scores.sort_values(['Score', 'Feat_names'], ascending=[False, True])

            self.features_name = feat_selector.get_feature_names_out(self.features_name)

            self.X, self.X_test = X_trans, X_test_trans

        elif self._feat_select_opt is None:
            return

        else:
            raise ValueError(f'Invalid feature selection option: {self._feat_select_opt}')

        return feat_selector

    def _add_lang(self):
        """
        Add a feature with the language of the post (en:0, fr:1)
        """
        if self._lang_id:
            # Train
            en_train_array = (self.words_dataset.train_data['lang'] == 'en').astype(int).to_numpy()  # 0: en,
            en_train_array = sp.csr_matrix(en_train_array).reshape(-1, 1)

            fr_train_array = (self.words_dataset.train_data['lang'] == 'fr').astype(
    int).to_numpy()  # 0: en,
            fr_train_array = sp.csr_matrix(fr_train_array).reshape(-1, 1)

            self.X = sp.csr_matrix(sp.hstack([self.X, en_train_array, fr_train_array])
    )
            self.features_name = np.append(self.features_name, ['is_en', 'is_fr'])

            # Test
            en_test_array = (self.words_dataset.test_data['lang'] == 'en').astype(int)
    .to_numpy()  # 0: en,
            en_test_array = sp.csr_matrix(en_test_array).reshape(-1, 1)

            fr_test_array = (self.words_dataset.test_data['lang'] == 'fr').astype(int)
    .to_numpy()  # 0: en,
            fr_test_array = sp.csr_matrix(fr_test_array).reshape(-1, 1)

            self.X_test = sp.csr_matrix(sp.hstack([self.X_test, en_test_array,
    fr_test_array]))

    def _normalize_data(self):
        """
        Remove mean and var of data
        """

        scaler = None
        if self._standardize_data:
            scaler = preprocessing.StandardScaler().fit(self.X.toarray())

            self.X = scaler.transform(self.X.toarray())
            self.X_test = scaler.transform(self.X_test.toarray())

        return scaler

    def get_params(self):
        return {
            # 'max_feat': self._max_feat,
            'n_gram': self._n_gram,
            'feat_type': self._feature_type,
            'lemmatized': self._lemmatize,
            'lang': self._lang_id,
            'standardized': self._standardize_data,
            'rm_accents': self._rm_accents,
            'feat_select': self._feat_select_opt,
            'n_feat': self._n_feat_select,
        }
```

7.4 cross_val_score.py

```
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold
import itertools

def cross_val_score(
    model_class, X, y, cv=5, base_params=None, cv_params=None, results_df=None,
    ds_name=None
):
    """To perform K-Fold validation to find the best combination of parameters for a
    given model.

    The parameters in `base_params` are kept the same for all tests.

    K-fold validation is performed for each combination of params in `model_params`.

    Args:
        model_class (model class): Model to test
        X (NDArray): Training dataset
        y (NDArray): Labels of the training dataset
        cv (int, optional): Number of folds. Defaults to 5.
        base_params (dict, optional): Keywords to pass to the model. Defaults to {}.
        model_params (dict, optional): Keywords to pass to the model. Defaults to {}.

    Returns:
        pd.DataFrame: Score of each combination
    """
    kf = KFold(n_splits=cv, shuffle=True)

    results = []

    # Find all combinations of parameters
    if cv_params is not None:
        keys, values = zip(*cv_params.items())
        all_combs = [dict(zip(keys, v)) for v in itertools.product(*values)]
    else:
        all_combs = [{}]

    # Find the best combination w/ CV
    for i, each_comb in enumerate(all_combs):
        print(f'\r\tCombination {i+1}/{len(all_combs)}', end='')
        model = model_class(**base_params, **each_comb)  # Create model with curr
    params combination
        # print(f"\tParams: {each_comb}", end='')

        # Check if model has already been trained on this ds
        if not results_df.empty:
            matching_row = results_df[
                (results_df['Model'].apply(type) == type(model))
                & (results_df['Params'] == each_comb)
                & (results_df['Dataset'] == ds_name)
            ]
            if not matching_row.empty:
                continue

        score = 0

        comb_ok = True  # Set to False if the combination of parameters is invalid

        for i, (train_idx, test_idx) in enumerate(kf.split(X)):
            if not comb_ok:
                break

            X_train = X[train_idx]
            X_test = X[test_idx]
            y_train = y[train_idx]
            y_test = y[test_idx]
            try:
                score += model.fit(X_train, y_train).score(X_test, y_test)

            except ValueError as err:
                comb_ok = False
                err_msg = err

        score /= cv

        if comb_ok:
            # Train on whole ds
            acc = model.fit(X, y).score(X, y)

            results.append({'Params': each_comb, 'Score': score, 'Model': model, 'Acc': acc})

        if not comb_ok:
            print(f"Invalid model: {err_msg}")

    return pd.DataFrame(results)
```