Rotten Tomatoes Reviews — Naive Bayes Classifier

Link to the Jupyter Notebook

The Rotten Tomatoes Reviews dataset is a compilation of movie reviews that were obtained from the well-known movie review website Rotten Tomatoes. The dataset consists of the reviews' text and a corresponding label that specifies whether the review was classified as "fresh" or "rotten", based on Rotten Tomatoes' proprietary review aggregation system.

This dataset is a highly valuable resource for individuals interested in conducting sentiment analysis and natural language processing, including researchers, data analysts, and machine learning practitioners. It contains reviews from a diverse group of critics and publications, encompassing a wide range of movies across various genres and languages.

Using a movie review, we have to predict if the movie is 'Fresh' or 'Rotten'. Let's dive into the problem.

. . .

> *About the dataset*

The dataset contains the label of two categories—Fresh/Rotten and the Reviews—user review given as a string. This dataset is a good benchmark for starting binary classification problems in the subject of Natual Langauge Processing models.

> *References*

Koushiki Dasgupta Chaudhuri—Published On March 21, 2022, and Last Modified On March 25th, 2022

https://www.analyticsvidhya.com/blog/2022/03/building-naive-bayes-classifier-from-scratch-to-perform-sentiment-analysis/

Eiki—January 3, 2019

https://medium.com/@eiki1212/natural-language-processing-naive-bayes-classification-in-python-e934365cf40c#:~:text=Bayes'%20theorem%20is%20a%20math,affect%20to%20the%20final%20probability.

. . .

> *Core Logic—Bayes Theorem*

Bayes theorem is a fundamental concept in probability theory that provides a way to calculate conditional probabilities. Naive Bayes is a machine learning algorithm that uses the Bayes theorem to predict the probability of a given sample belonging to a particular class based on its features. It assumes that the features are independent of each other, which is why it's called "naive."

Bayes theorem can be represented using the following formula:

$$P(A|B) = \frac{P(B|A)\ P(A)}{P(B)}$$

Bayes Theorem

where P(A|B) is the probability of event A given that event B has occurred, P(B|A) is the probability of event B given that event A has occurred, P(A) is the prior probability of event A, and P(B) is the prior probability of event B.

Advantages of Naive Bayes Classifier —

1. *Simplicity*: Naive Bayes is a simple and easy-to-understand algorithm that is easy to implement and works well for many classification problems.
2. *Efficiency*: It can be very fast and efficient, making it suitable for large datasets and real-time applications.
3. *Good performance with small sample sizes*: Naive Bayes can still perform well even when the dataset is small, which is a common scenario in many real-world applications.
4. *Can handle both categorical and numerical data*: It can work with both categorical and numerical data, making it versatile and useful in many different types of classification problems.
5. *Can handle high-dimensional data*: Naive Bayes can handle a large number of features or variables, making it suitable for high-dimensional data.

Disadvantages of Naive Bayes Classifier —

1. *Naive assumptions*: The Naive Bayes classifier assumes that all features are independent of each other, which is not always the case in real-world scenarios. This oversimplified assumption can lead to inaccurate predictions.
2. *Zero-frequency problem*: If a feature has not been seen in the training data for a particular class, then the Naive Bayes classifier assigns it a probability of zero. This can lead to incorrect predictions if the feature is present in the test data.
3. *Limited expressiveness*: Naive Bayes is a probabilistic model that works well for simple classification problems, but it may not be expressive enough for complex classification tasks.
4. *Imbalanced data*: Naive Bayes can be biased towards the majority class in imbalanced datasets, which can result in poor performance for the minority class.

. . .

Execution

Libraries Used

```
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict

import nltk
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('omw-1.4')
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
```

I will try to use the minimum from the libraries and do most of it from scratch.

However, libraries like Numpy and NLTK can be highly useful for handling mathematical and language processing tasks respectively.

Load the data

```
def read_data(filename='./data/rt_reviews.csv'):
    """
    Load the dataset

    Parameters
    ----------
    filename - string

    Returns
    -------
    reviews - numpy array of strings : feedback
    labels - numpy array of strings : fresh / ro
    """
    reviews = []
    labels = []
    with open(filename, 'r', encoding='latin-1')
        for line in f:
            line = line.split(',')
            label, review = line[0], ''.join(lin

            labels.append(label)
            reviews.append(review)

    ## returning from 1st index; 1st line is jus
    return np.array(labels[1:]), np.array(review
```

Read the dataset given as a CSV file.

Note that I am using Latin-1 encoding to read the CSV file deliberately because while using the default encoding, the CSV reader could not convert a few bytes.

This function returns the labels and reviews as numpy arrays.

```
labels, reviews = read_data()
labels.shape, reviews.shape
```

```
((480000,), (480000,))
```

> *Split the Dataset*

I will split the dataset into 3 sets—training set, testing set, and validation set.

```
def train_test_val_split(labels, reviews):
    """
    splits the dataset into training, testing an

    Parameters
    ----------
    labels - numpy array of strings : fresh / ro
    reviews - numpy array of strings : feedback


    Returns
    -------
    train_data - numpy array of 70% of reviews c
    train_labels - numpy array of 70% of labels
    test_data - numpy array of 20% of reviews ch
    test_labels - numpy array of 20% of labels c
    val_data - numpy array of 10% of reviews cho
    val_labels - numpy array of 10% of labels ch
    """
    ## assume that labels and reviews are numpy
    data_size = len(labels)

    ## shuffle the indices of the data
    shuffled_indices = np.random.RandomState(see

    ## split the indices into train, validation,
    train_indices = shuffled_indices[:int(0.7 *
    val_indices = shuffled_indices[int(0.7 * dat
    test_indices = shuffled_indices[int(0.8 * da

    ## use the indices to extract the correspond
    train_data = reviews[train_indices]
    train_labels = labels[train_indices]

    val_data = reviews[val_indices]
    val_labels = labels[val_indices]

    test_data = reviews[test_indices]
    test_labels = labels[test_indices]

    return train_data, train_labels, test_data,

train_data, train_labels, test_data, test_labels
```

This function has been hardcoded to divide the dataset into 70:20:10 for training, testing and validation respectively.

Note that I have set the seed for ***numpy.random*** *to 21 in order to reproduce the same random dataset for analysis.*

. . .

> *Data Exploration and Preparation*

```
def get_data_dist(labels):
    """
    prints the distribution of labels in a datas

    Parameters
    ----------
    labels - numpy array of labels
    """
    # assume that train_labels is a numpy array
    unique_labels, label_counts = np.unique(labe

    for label, count in zip(unique_labels, label
        print(f"- {label} : {round(100*count/len

    return None
```

This function returns the distribution of data in each class for a given dataset. We can observer the data distribution in all the datasets below -

```
for name, labs in zip(['Train','Test','Validatio
    print(f'\nClass distribution in {name} datas
    get_data_dist(labs)
```

```
Class distribution in Train dataset
- fresh : 50.04%
- rotten : 49.96%

Class distribution in Test dataset
- fresh : 49.75%
- rotten : 50.25%

Class distribution in Validation dataset
- fresh : 50.2%
- rotten : 49.8%
```

> *Data Preprocessing*

As we have the text dataset, It needs heavy cleaning because people can write anything in the reviews including empty spaces, inverted commas etc. which are meaningless in this context.

```
def clean_data(data):
    """
    clean data from new line character, empty sp

    Parameters
    ----------
    data - numpy array of strings

    Returns
    -------
    data - numpy array of strings
    """
    data = np.array([i.replace('"','').replace('
    return data
```

You can observe the difference in the data below -

```
----------------------------- Training data befo
['" Gloriously daft but with a good deal of hear
 '" The Back-Up Plan represents a major comeback
 '" The acting can be so-so the story implausibl
 ...
 '" The film is indeed a bit pat. Sweet and funn
 ' More concerned with recruiting the testostero
 '" It strives awfully hard for depth but more o
```

```
----------------------------- Training data afte
["Gloriously daft but with a good deal of heart
 "The Back-Up Plan represents a major comeback f
 'The acting can be so-so the story implausible
 ...
 "The film is indeed a bit pat. Sweet and funny
 'More concerned with recruiting the testosteron
 'It strives awfully hard for depth but more oft
```

Stopwords

Stopwords are common words that are often removed from text data during natural language processing (NLP) tasks. These are words that occur frequently in a language and are not considered to have much meaning or context on their own, such as “the”, “and”, “is”, “of”, “in”, and so on.

Stopwords are important in NLP because they can have a negative impact on the accuracy and efficiency of many text analysis tasks, such as sentiment analysis, topic modeling, and *document classification*. Removing stopwords can help to improve the quality of these tasks by reducing the noise and complexity of the data. By eliminating words that do not provide much context or meaning, the remaining words in the text can be more informative and useful for analysis.

In addition to improving the accuracy and efficiency of text analysis, removing stopwords can also help to reduce the amount of memory and storage required for processing large amounts of text data, as well as reducing the computational time required for processing.

```
stop_words = ['i', 'me', 'my', 'myself', 'we', '
              "you're", "you've", "you'll", "you
              'yourselves', 'he', 'him', 'his',
              'hers', 'herself', 'it', "it's", '
              'their', 'theirs', 'themselves', '
              'this', 'that', "that'll", 'these'
              'were', 'be', 'been', 'being', 'ha
              'does', 'did', 'doing', 'a', 'an',
              'because', 'as', 'until', 'while',
              'about', 'against', 'between', 'in
              'after', 'above', 'below', 'to', '
              'on', 'off', 'over', 'under', 'aga
              'here', 'there', 'when', 'where',
              'each', 'few', 'more', 'most', 'ot
              'only', 'own', 'same', 'so', 'than
              'can', 'will', 'just', 'don', 'now
              'd', 'll', 'm', 'o', 're', 've', '
              'didn', "didn't", 'doesn', "doesn'
              "hadn't", 'hasn', "hasn't", 'haven
              'mightn', "mightn't", 'mustn', "mu
              'wasn', "wasn't", 'weren', "weren'
```

The preprocessing pipeline looks following for our use-case :

```
def preprocess(text):
    """
    steps like -
    converts all characters to lower
    splits sentence to words
    removes non-alphabetical words

    Parameters
    ----------
    text - string

    Returns
    -------
    text - string (preprocessed)
    """
    text = text.lower()
    ## word_tokenize splits a sentence into word
    text = word_tokenize(text)
    text = [word for word in text if (word.isalp
    text = ' '.join(text)

    return text
```

We will preprocess any incoming data before using it for training or prediction.

```
## preprocess all the datasets
train_data = np.array([preprocess(i) for i in tr
test_data = np.array([preprocess(i) for i in tes
val_data = np.array([preprocess(i) for i in val_
```

> *Create Vocabulary*

In our context of document classification, we need to create the vocabulary of words in the documents to calculate the probabilities for prediction.

```
## lemmatizer instance to keep words with same m
## for example - converts 'went' to 'go'
lemm = WordNetLemmatizer()

## initialize dictionaries for vocabulary
vocab_all = defaultdict(int)
vocab_fresh = defaultdict(int)
vocab_rotten = defaultdict(int)

for ind, item in enumerate(train_data):
    words_unq = []

    ## split sentence by words linguistically
    words = nltk.word_tokenize(item)

    for word in words:
        ## lemmatize words to create a good voca
        words_unq.append(lemm.lemmatize(word, po
        words_unq.append(word)

    ## only keep unique set of words, remove dup
    words_unq = set(words_unq)

    if train_labels[ind] == 'fresh':
        for word in words:
            vocab_all[word] += 1
            vocab_fresh[word] += 1
    else:
        for word in words:
            vocab_all[word] += 1
            vocab_rotten[word] += 1

vocab_all = {key:value for key, value in vocab_a
vocab_fresh = {key:value for key, value in vocab
vocab_fresh = {key:value for key, value in vocab
```

Removing the words which are not occuring very frequently, as they do not really affect the accuracy and consume processing power.

NOTE—I am using a lemmatizer.

What is lemmatization

Lemmatization is the process of reducing words to their base or dictionary form, known as the lemma. This is achieved by removing

inflectional endings and considering the context and part of speech of the word. The goal of lemmatization is to reduce different forms of a word to a common base form, making it easier to analyze and compare words in natural language processing (NLP) tasks.

for example—converts ‘went’ to ‘go’; because both the words share the context.

. . .

> *Model Execution*

```
## find the counts of fresh and rotten labels in
counts_fresh, counts_rotten = np.unique(train_la
print(np.unique(train_labels, return_counts=True
```

Counting the number of labels in each class to use later in calculating prediction probabilities.

```
def predict(data):
    """
    predicts the label for a review using naive

    Parameters
    ----------
    data - string : movie review

    Returns
    -------
    label - string : fresh/rotten
    """
    ## define the local constants
    prob_fresh, prob_rotten, label = 1, 1, None

    base1 = (len(vocab_all) + counts_fresh)
    base2 = (len(vocab_all) + counts_rotten)

    for word in data.split():
        if word in vocab_fresh:
            prob_fresh *= (vocab_fresh[word] / b
        else:
            prob_fresh /= base1

        if word in vocab_rotten:
            prob_rotten *= (vocab_rotten[word] /
        else:
            prob_rotten /= base2

    if prob_fresh > prob_rotten:
        label = 'fresh'
    else:
        label = 'rotten'

    return label
```

Problem with this approach —

This code is prone to the *zero-frequency problem* as discussed above. In naive Bayes document classification, this problem arises when a feature is absent in the training dataset for a particular class, resulting in a conditional probability of zero. This can lead to incorrect predictions for the test data that contain the missing feature. Various techniques such as Laplace smoothing and add-k smoothing can be used to address this problem by adding a small value to the probability estimate for unseen features.

To solve this -

```
def predict_with_smoothing(data):
    """
    predicts the label for a review using naive

    Parameters
    ----------
    data - string : movie review

    Returns
    -------
    label - string : fresh/rotten
    """
    ## define the local constants
    prob_fresh, prob_rotten, label = 1, 1, None

    base1 = (len(vocab_all) + counts_fresh)
    base2 = (len(vocab_all) + counts_rotten)

    for word in data.split():
        if word in vocab_fresh:
            prob_fresh *= ((vocab_fresh[word] +
        else:
            prob_fresh /= (base1 + 1)

        if word in vocab_rotten:
            prob_rotten *= ((vocab_rotten[word]
        else:
            prob_rotten /= (base2 + 1)

    if prob_fresh > prob_rotten:
        label = 'fresh'
    else:
        label = 'rotten'

    return label
```

Smoothing —

One of the solutions to the zero frequency problem is to use smoothing techniques such as Laplace smoothing or add-k smoothing, which add a small value to the probability estimate for unseen features.

In this case—the value of k = 1.

. . .

Performance Analysis

```
def calc_accuracy(pred, actual):
    """
    return prediction accuracy

    pred - list of predicted labels
    actual - list of actual labels
    """

    accuracy = round(100*np.mean(pred == actual)
    return accuracy
```

We can calculate the accuracy by this custom function and see if the model is any good.

```
print(f'Accuracy for Training Dataset - {calc_ac
print(f'Accuracy for Testing Dataset - {calc_acc
print(f'Accuracy for Validation Dataset - {calc_
```

```
Accuracy for Training Dataset - 81.54%
Accuracy for Testing Dataset - 79.25%
Accuracy for Validation Dataset - 79.21%
```

Here, the model seems to working pretty well. This can be surely improved by advancing the preprocessing steps and improving the smoothing algorithm.

Another solution is to use more advanced techniques such as n-grams or term frequency-inverse document frequency (TF-IDF) to capture the context and importance of words in text data. This technique helps in getting better accuracy in such context of problems.

. . .

Probability of THE occurence

The probability of the word “the” occurring in all the documents can be derived by -

P[“the”] = num of documents containing ‘the’ / num of all documents

```
labels, reviews = read_data()

print(round(100*sum([True for i in reviews if 't
```

This gives an output of **63%** which means that the probability of the word “the” occurring in all the documents is *0.63*

> *Conditional probability based on the sentiment*

Similarly, we can calculate the conditional probability of the occurrence of “the” with respect to positive reviews.

P[“the” | Positive] = No. of positive documents containing “the” / num of all positive review documents

We can find the positive review documents easily by -

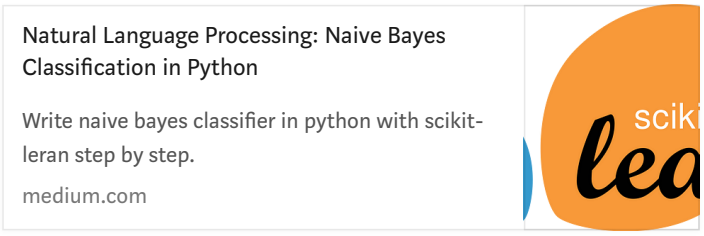
```
pos_doc = reviews[labels=='fresh']

round(100*sum([True for i in pos_doc if 'the' in
```

This gives the answer—**63.56%** which means the probability of “the” occurring given that it is a positive review is **0.6356**

. . .

> *Contribution*

For this project, I have used a few articles as a reference. Most of the context I got about Naive Bayes algorithm and text classification is from -



My key contributions to this project was -

- Defining the stopwords—When I used the generic stopwords, the accuracy was around 75%. But I thought that in the context of positive-negative text classification, some words like not, shouldn't, wouldn't etc. can play an important role.
- This small factor actually bumped my accuracy up by 4 percent.
- I am only processing alphabets, not even alpha numerics.
- Doing lemmatization after creating vocabulary and before that actually affected the accuracy for some reason.

> *Technical Challenges*

- One of the challenges was to find out what should be the value of k in add-k method to solve zero frequency problem.

- Sometimes, for smaller k values, the smoothing technique does not yield great results.

> *Conclusion*

The Rotten Tomatoes Reviews is an amazing dataset for beginners to explore the Naive Bayes Algorithm for text classification. The key takeaways from this exercise were -

- The Naive Bayes is a highly powerful algorithm.
- Text classification is significantly affected by the choice of words you ignore i.e. stopwords.
- Techniques like lemmatization, stemming improves the vocabulary and probabilities significantly.
- Zero Frequency Problem is one of the classic issues in text classification and can be solved by a few smart work-arounds. Simplest of those is add-k method.
- This exercise also taught about the need and importance of the TF-IDF technique which captures the context information.