# Introduction to Artificial Intelligence HW#3

Name: LeeJaeJun

Student ID: 2021313030

## Abstract

This project focuses on implementing a Naive Bayes Classifier (NBC) for sentiment analysis using a dataset containing reviews and their respective star ratings. The classifier employs Laplace smoothing to enhance the effectiveness of the Naive Bayes algorithm. The primary goal is to predict the sentiment of reviews based on the words they contain and to analyze the efficiency of the model. Additionally, the impact of varying the amount of training data on the model's performance is examined. The results demonstrate the classifier's ability to accurately predict sentiments and provide insights into the relationship between training data size and model efficacy.

## Definition

**Naive Bayes Classifier**: A probabilistic machine learning model used for classification tasks. It is based on Bayes' Theorem and assumes independence between the features.

**Laplace smoothing**: A technique used to handle the problem of zero probabilities in probabilistic models. For an attribute $Xi$ with $k$ possible values, Laplace smoothing adds 1 to the numerator and $k$ to the denominator of the maximum likelihood estimate. This adjustment ensures that no probability is ever exactly zero, which helps in making the model more robust.

**Stop word**: Words in a language that are often filtered out before processing textual data. These words (e.g., "is," "the," "at") are considered to have little value in terms of information retrieval and text analysis, as they appear frequently across different texts without contributing significantly to the sentiment or main content.

**Special characters:** punctuation marks and symbols that are not part of the standard alphanumeric set.

**Evaluation measure:**

$$Accuracy = \frac{TrueNegatives + TruePositive}{TruePositive + FalsePositive + TrueNegative + FalseNegative}$$

## Methodology

### Preprocessing

- Convert all text to lowercase

- **Special Characters**: The special characters list was created based on a provided reference site and includes characters such as " !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~". Additionally, numbers were removed from the reviews as they do not contribute to determining the sentiment.

- **Stop Words**: The stop words are based on the words listed in the `stopwords.txt` file. However, this file does not include certain contractions and auxiliary verbs. Therefore, additional stop words were included without altering the scope of the basic words in the `stopwords.txt` file. For example, while "cannot" is included in the file, "can" is not, and "What" and "Where" are included, but "Who" is missing. These gaps were filled by adding the necessary stop words manually to ensure they do not appear among the top 1000 most frequent words used in the NBC model.

- **Contractions**: For certain contractions, removing the special character (e.g., the apostrophe) could result in different words with distinct meanings (e.g., "I'll" becomes "ill", "I'd" becomes "id"). These were removed before other special characters were processed.

- Spacing: To ensure proper tokenization, double spaces were replaced with single spaces.

### Sentiment Determination

Following the assignment guidelines, a review with a 5-star rating is considered positive, while all other ratings are considered negative.

### Laplace Smoothing

Laplace smoothing was applied under the assumption that each of the top 1000 selected words appears at least once in the dataset. This technique adds 1 to the numerator and the number of unique words (k) to the denominator to avoid zero probabilities.

### Accuracy Evaluation

Accuracy was calculated based on the number of correct predictions out of the total predictions, without differentiating between false positives, false negatives, true positives, or true negatives. This aligns with the standard definition of accuracy: the ratio of correctly predicted instances to the total instances.

## Code Details

### NBC.py

```
import csv
import re
import heapq

class NaiveBayesClassifier:
    def __init__(self):
        """
        Initialize the NaiveBayesClassifier class.
        """
        # Initialize instance variables
        self.train_data = []  # List to store training data
        self.test_data = []  # List to store test data
        self.frequent_words = []  # List to store frequent words
        self.word_positive_probability = {}  # Dictionary to store the probability
of each word in positive reviews
        self.word_negative_probability = {}  # Dictionary to store the probability
of each word in negative reviews
        self.positive_probability = 0  # Probability of a positive review
        self.negative_probability = 0  # Probability of a negative review

        # Load stopwords from file and add additional contractions and common
words
        with open("data/stopwords.txt", "r") as file:
            # Read stopwords from file and remove leading/trailing whitespaces
            self.stopwords = [word.strip() for word in file.readlines()]

            # Add additional contractions and common words to the stopwords list
            additional_stopwords = [
                "will", "can", "im", "ive", "dont", "am", "didnt", "who", "being",
"youre", "wont", "doesnt", "may",
                "theres", "wouldnt", "isnt", "theyre", "havent", "youll",
"werent", "arent", "weve", "hes", "youve",
                "whats", "hadnt", "shouldnt", "youd", "wasnt"
            ]
            self.stopwords.extend(additional_stopwords)

    def read_data(self, filename, mode = "train"):
        """
        Read data from a CSV file.

        Args:
            filename (str): The name of the CSV file.
            mode (str, optional): The mode of reading data. "train" or "test" mode
is supported. Defaults to "train".
        """
        # Read data from CSV file
        with open(filename, "r", encoding="utf8") as file:
```

```
            reader = csv.reader(file)
            next(reader)  # skip header
            data = list(reader)

            # Store the read data based on the mode
            if mode == "train":
                self.train_data = data  # Store the data for training
            elif mode == "test":
                self.test_data = data  # Store the data for testing

    def preprocess(self, mode = "train"):
        """
        Preprocess the data: lowercase, remove special characters, numbers, and
stopwords.

        Args:
            mode (str, optional): The mode of preprocessing data. "train" or
"test" mode is supported. Defaults to "train".
        """
        # Select the data based on the mode
        if mode == "train":
            data = self.train_data
        elif mode == "test":
            data = self.test_data

        # Return if no data is available
        if not data:
            return

        # Preprocess the data
        for review in range(len(data)):
            # Lowercase the text
            lower_text = data[review][1].lower()

            # Remove contractions and special characters
            lower_text = lower_text.replace("i'll", "")
            lower_text = lower_text.replace("i'd", "")
            special_characters_removed_text =
re.sub(r'[!"#$%&\'()*+,-./:;<=>?@\[\\\]^_`{|}~]', '', lower_text)

            # Remove numbers
            number_removed_text = re.sub(r'\d+', '',
special_characters_removed_text)

            # Replace double spaces to single space
            double_spaces_removed_text = re.sub(r'\s{2,}', ' ',
number_removed_text)

            # Split the text into words
            words = double_spaces_removed_text.split()

            # Remove stopwords
```

```
            words_without_stopwords = [word for word in words if word not in
self.stopwords]

            # Update the label
            data[review][0] = 1 if data[review][0] == '5' else 0

            # Update the preprocessed text
            data[review][1] = words_without_stopwords

        # Store the preprocessed data based on the mode
        if mode == "train":
            self.train_data = data
        elif mode == "test":
            self.test_data = data

    def get_frequent_words(self):
        """
        Get the most frequent words from the training data.

        This function counts the occurrences of each word in the training data and
        stores the top 1000 words in the `frequent_words` attribute. It then
prints
        the top 50 frequent words.

        Returns:
            None
        """
        # Return if no training data is available
        if not self.train_data:
            return

        # Count the occurrences of each word in the training data
        frequent_words = {}
        for review in range(len(self.train_data)):
            for word in self.train_data[review][1]:
                if word not in frequent_words:
                    frequent_words[word] = 1
                else:
                    frequent_words[word] += 1

        # Store the top 1000 frequent words
        self.frequent_words = [word for word, _ in heapq.nlargest(1000,
frequent_words.items(), key=lambda item: item[1])]

        # Print the top 50 frequent words
        print("Top 50 Frequent Words: ")
        for word in self.frequent_words[:50]:
            print(word)

    def train(self, percentage):
        """
        Train the model using a specified percentage of the training data.
```

```
        Args:
            percentage (float): The percentage of the training data to use for
training.
        """
        # Select the required portion of the training data
        data = self.train_data[:int(len(self.train_data) * (percentage / 100))]

        # Initialize counters for positive and negative reviews and words
considering Laplace smoothing
        positive_word_count = [1] * len(self.frequent_words)
        negative_word_count = [1] * len(self.frequent_words)
        positive_reviews = 0
        negative_reviews = 0

        # Iterate over the selected data
        for review in range(len(data)):
            # If the review is positive
            if data[review][0] == 1:
                positive_reviews += 1
                # Update the word counts for positive reviews
                for word in data[review][1]:
                    if word in self.frequent_words:
                        index = self.frequent_words.index(word)
                        positive_word_count[index] += 1
            # If the review is negative
            else:
                negative_reviews += 1
                # Update the word counts for negative reviews
                for word in data[review][1]:
                    if word in self.frequent_words:
                        index = self.frequent_words.index(word)
                        negative_word_count[index] += 1

        # Calculate the word probabilities for positive and negative reviews
considering Laplace smoothing
        self.word_positive_probability = {
            word: positive_word_count[self.frequent_words.index(word)] /
            (positive_reviews + len(self.frequent_words)) for word in
self.frequent_words
        }
        self.word_negative_probability = {
            word: negative_word_count[self.frequent_words.index(word)] /
            (negative_reviews + len(self.frequent_words)) for word in
self.frequent_words
        }

        # Calculate the class probabilities for positive and negative reviews
        self.positive_probability = positive_reviews / len(self.train_data)
        self.negative_probability = negative_reviews / len(self.train_data)

    def predict(self, text):
```

```
        """
        Predict the label of a given text.

        Args:
            text (list): The text to be predicted.

        Returns:
            int: The predicted label (1 for positive, 0 for negative).
        """
        # Initialize the probabilities with the class probabilities
        positive = self.positive_probability
        negative = self.negative_probability

        # Multiply the class probabilities by the word probabilities for each word
in the text
        for word in text:
            # Check if the word is in the positive word probabilities
            if word in self.word_positive_probability:
                positive *= self.word_positive_probability[word]

            # Check if the word is in the negative word probabilities
            if word in self.word_negative_probability:
                negative *= self.word_negative_probability[word]

        # Return the predicted label based on the comparison of the positive and
negative probabilities
        if positive > negative:
            return 1  # Positive label
        else:
            return 0  # Negative label

    def evaluate(self):
        """
        Evaluate the performance of the Naive Bayes Classifier on the test data.

        Returns:
            float: The accuracy of the classifier.
        """
        # Predict the labels for the test data
        prediction = [self.predict(review[1]) for review in self.test_data]

        # Get the correct labels for the test data
        correct_answer = [review[0] for review in self.test_data]

        # Initialize counters for prediction success and failure
        prediction_success = 0
        prediction_failure = 0

        # Iterate over the predictions and correct answers and count the number of
correct predictions
        for val1, val2 in zip(prediction, correct_answer):
            if val1 == val2:
```

```
                prediction_success += 1
            else:
                prediction_failure += 1

        # Calculate the accuracy of the classifier
        accuracy = prediction_success / (prediction_success + prediction_failure)

        return accuracy
```

**NaiveBayesClassifier** class implements the Naïve Bayes algorithm for text classification. It is designed to classify text data into two classes: positive and negative. This model is trained using a specified amount of training data and its accuracy is evaluated on a test dataset.

**- __intit__(self)**

Initializes the NaiveBayesClassifier class by setting up instance variables for storing training and test data, frequent words, word probabilities, and class probabilities. It also loads stop words from a file and includes additional contractions and common words to ensure comprehensive filtering.

**- read_data(self, filename, mode='train')**

Reads data from a CSV file and stores it in either the train_data or test_data attribute based on the provided mode. This method facilitates the separation of training and testing datasets.

**- prerprocess(self, mode='train')**

Preprocesses the data by converting text to lowercase, removing special characters, numbers, and stop words. The cleaned data is then stored back in either the train_data or test_data attribute, ensuring consistency for model training and evaluation.

**- get_frequent_words(self)**

Counts the frequency of each word in the training data and stores the top 1000 most frequent words in the frequent_words attribute. It also prints the top 50 frequent words to provide insight into the most common terms in the dataset.

**- train(self, percentage)**

Trains the model using a specified percentage of the training data. It calculates the word probabilities for positive and negative reviews using Laplace smoothing and stores these probabilities in the word_positive_probability and word_negative_probability attributes. Additionally, it calculates the overall class probabilities for positive and negative reviews.

- **predict(self, text)**

Predicts the sentiment of a given text by multiplying the class probabilities with the word probabilities for each word in the text. It returns the predicted label, where 1 indicates a positive sentiment and 0 indicates a negative sentiment.

- **evaluate(self)**

Evaluates the performance of the Naive Bayes Classifier on the test data. It predicts the labels for the test dataset and compares them with the actual labels. The accuracy of the classifier is then calculated and returned, providing a measure of the model's performance.

### draw.py

```
import matplotlib.pyplot as plt


def drawPlot(dict):
    """
    This function is used to draw a plot of the model accuracy with varying
training data sizes.

    Parameters:
    dict (dict): A dictionary where the keys are the training data sizes and the
values are the corresponding model accuracies.
    """
    # Get the keys and values from the dictionary
    amount_of_training_data = list(dict.keys())
    accuracy = list(dict.values())

    # Draw the plot
    plt.plot(amount_of_training_data, accuracy)  # Plot the model accuracy
    plt.scatter(amount_of_training_data, accuracy, color='r', zorder=5)  # Scatter
plot for clear visualization

    # Set the x-axis ticks
    plt.xticks(range(0, 110, 10))

    # Add labels and titles
    plt.title("Model Accuracy with Varying Training Data Sizes")  # Title of the
plot
    plt.xlabel("Amount of training data (%)")  # Label for the x-axis
    plt.ylabel("Accuracy")  # Label for the y-axis

    # Add accuracy values as text on the plot
    for index, (x, y) in enumerate(zip(amount_of_training_data, accuracy)):
        plt.text(x, y + 0.0015, f'{y:.4f}', fontsize=9, ha='right')

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

This code defines a function drawPlot that takes a dictionary as input. The keys of the dictionary represent the training data sizes, and the values represent the corresponding model accuracies.

The function uses the matplotlib library to create a plot of the model accuracy with varying training data sizes. It plots the accuracy using plt.plot and adds a scatter plot for clear visualization using plt.scatter. The x-axis ticks are set using plt.xticks. The plot is labeled with a title, x-axis label, and y-axis label using plt.title, plt.xlabel, and plt.ylabel respectively. The accuracy values are added as text on the plot using a loop that iterates over the keys and values of the input dictionary. Finally, the plot is displayed using plt.show.

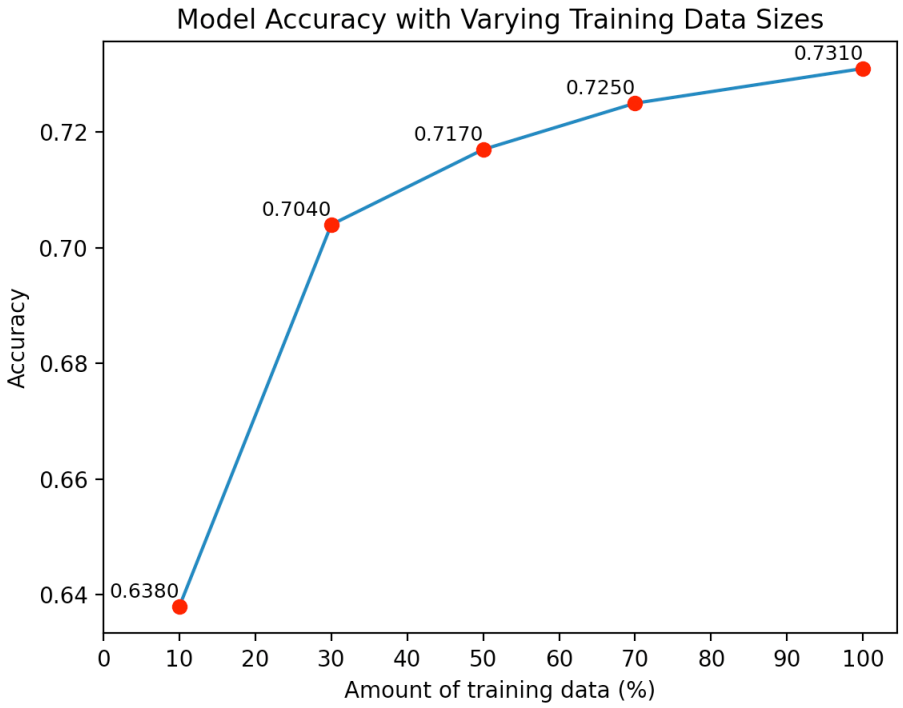
### main.py

```python
from NBC import NaiveBayesClassifier
from draw import drawPlot

if __name__ == "__main__":
    tran_file_path = "data/train.csv"
    test_file_path = "data/test.csv"

    NaiveBayesClassifier = NaiveBayesClassifier()

    NaiveBayesClassifier.read_data(tran_file_path, "train")
    NaiveBayesClassifier.preprocess("train")
    NaiveBayesClassifier.get_frequent_words()

    NaiveBayesClassifier.read_data(test_file_path, "test")
    NaiveBayesClassifier.preprocess("test")

    percentages = [10, 30, 50, 70, 100]
    accuracy = {}

    # Train and evaluate the model for each percentage
    for percentage in percentages:
        NaiveBayesClassifier.train(percentage)
        accuracy[percentage] = NaiveBayesClassifier.evaluate()

    drawPlot(accuracy)
```

This file defines the NaiveBayesClassifier class and performs training and testing, followed by visualization using the drawPlot function. The training data size is progressively increased to 10%, 30%, 50%, 70%, and 100%, and the corresponding accuracy values are obtained and visualized.

## Results and Discussion

**Top 50 Frequent Words**: food, place, just, like, here, time, back, great, service, only, went, know, people, well, best, order, told, going, first, love, minutes, staff, ordered, down, now, way, chicken, day, came, restaurant, nice, car, see, still, asked, take, little, try, store, want, experience, made, right, friendly, new, think, bad, took, come, bar.




### 1. 10% Training Data

When only 10% of the training data is used, the model's accuracy is approximately 0.638. This is the lowest accuracy observed in the graph, indicating that the model is not very effective with such a small amount of training data.

### 2. 30% Training Data

With 30% of the training data, the accuracy jumps significantly to around 0.704. This indicates that adding more training data helps the model learn better and improve its performance.

### 3. 50% Training Data

As the training data increases to 50%, the accuracy slightly increases to around 0.7179. This suggests diminishing returns; while more data improves the model, the rate of improvement slows down.

**4. 70% Training Data**

Like the previous cases, the accuracy increased slightly, but the rate of improvement decreased even further.

**5. 100% Training Data**

Using the full training dataset (100%), the accuracy reaches approximately 0.731. This is the highest accuracy observed, indicating that the model performs best when all available training data is used. However, the improvement from 70% to 100% is minimal, reinforcing the idea of diminishing returns.

The model's accuracy improves as the amount of training data increases, with the most significant gains observed when moving from 10% to 30% of the training data. Beyond 30%, the improvements become more incremental, indicating that while more data is generally beneficial, there are diminishing returns. Initial increases in data lead to substantial improvements, but the gains taper off as the dataset becomes larger.

Currently, the entire dataset size is small, resulting in fast computation speeds. However, when dealing with larger datasets that require longer training times, it may be more efficient to consider the optimal size of the dataset rather than using the entire dataset. This consideration is based on the observation that as the dataset size increases, the rate of accuracy improvement diminishes.

Using the entire dataset yields an accuracy of approximately 73%. To achieve higher accuracy, several improvements could be made. Firstly, during training and testing, many words are recognized as different due to grammatical variations. For example, "want" and "wanting" are treated as different words, as are "order" and "ordered." Additionally, singular and plural forms, such as those ending in -s or -es, are counted separately despite having the same meaning. Irregular forms and other grammatical variations also contribute to this issue. By recognizing these variations as the same word during training, the model's accuracy could be improved.

Moreover, for cases like "i'll" and "ill," where removing the special character (') results in completely different words, not handling these separately can lead to misinterpretations, such as treating "i'll" as "ill," thus distorting the results. Therefore, these words should be handled separately before removing the special character. Implementing these additional preprocessing steps would likely result in higher model accuracy.

Additionally, considering that this dataset is used for evaluating the sentiment of reviews, further improvements could be made by leveraging the specific characteristics of the data.

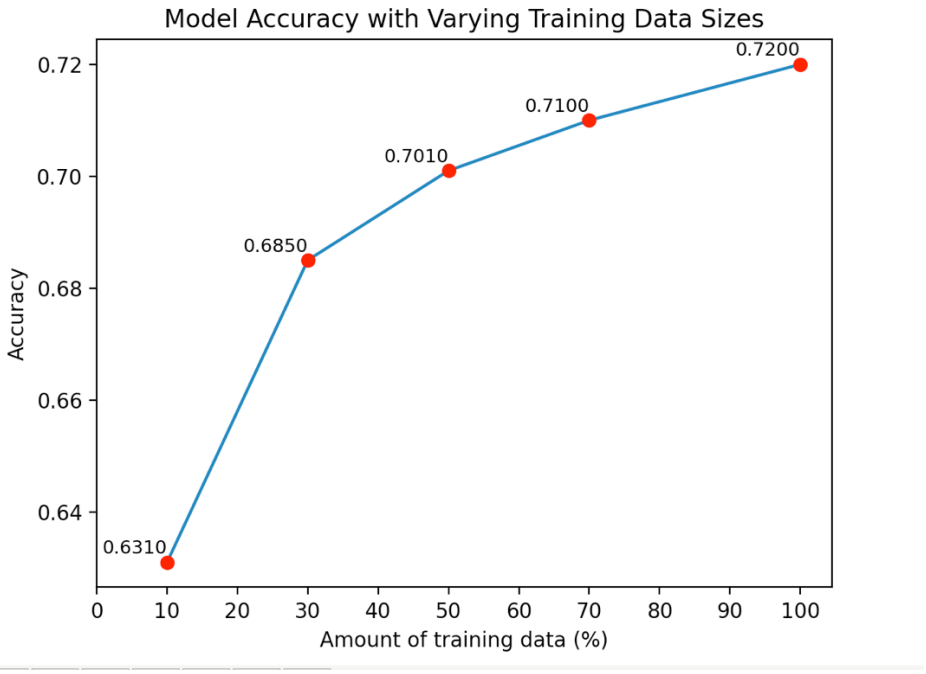
Removing less meaningful words and those that can be interpreted as both positive and negative depending on the context could enhance the model. Moreover, instead of processing single words, enhancing the model to consider phrases or sentences to capture additional context would likely result in a more accurate model.

### In the case of using only stopwords.txt

```
75          # lower_text = lower_text.replace("i'll", "")
76          # lower_text = lower_text.replace("i'd", "")
```

```
25          # additional_stopwords = [
26          #     "will", "can", "im", "ive", "dont", "am", "didnt", "who", "being", "youre", "wont", "doesnt", "may",
27          #     "theres", "wouldnt", "isnt", "theyre", "havent", "youll", "werent", "arent", "weve", "hes", "youve",
28          #     "whats", "hadnt", "shouldnt", "youd", "wasnt"
29          # ]
30          # self.stopwords.extend(additional_stopwords)
```

Top 50 Frequent Words: food, place, just, like, here, time, back, great, service, will, can, only, dont, im, ive, went, know, people, well, am, best, didnt, order, told, who, going, first, love, minutes, staff, ordered, down, now, way, chicken, day, came, restaurant, nice, car, see, still, asked, take, little, try, store, want, experience, made




This result is derived from preprocessing using only the words from stopwords.txt, without any additional preprocessing, unlike the initial results provided. It can be observed that the accuracy has decreased due to the inclusion of be-verbs and meaningless contractions. This indicates that applying the additional preprocessing steps would likely result in higher accuracy.

## References


- CS 188 Spring 2024. (2024). *Introduction to Artificial Intelligence*. University of California, Berkeley.

- Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

- OWASP Foundation. "Password Special Characters." OWASP, https://owasp.org/www-community/password-special-characters. Accessed 1 June 2024.

- Swami, Rohit. "Stopwords." Kaggle, https://www.kaggle.com/datasets/rowhitswami/stopwords. Accessed 1 June 2024.

- D'Auria, Erika. "Accuracy, Recall, Precision." Medium, https://medium.com/@erika.dauria/accuracy-recall-precision-80a5b6cbd28d. Accessed 1 June 2024.