Introduction to Artificial Intelligence HW#3

Name: LeeJaeJun

Student ID: 2021313030

Abstract

This project focuses on implementing the Naive Bayes Classifier (NBC) for sentiment analysis using a dataset containing reviews and star ratings. The classifier uses Laplace smoothing to increase the efficiency of the Naive Bayes algorithm. The main goal is to predict the attitude of the review based on the words contained in the review and to analyze the efficiency of the model. We also investigate the impact on the performance of the model with the amount of training data. The resulting classifier shows that it can accurately predict attitude about the review and provide insight 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 features.

Laplace smoothing: A technique used to solve the problem of zero probability 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 words: Words in a language that are often filtered out before processing text data. These words (e.g., "is", "the", "at") are considered of little value in terms of information retrieval and text analysis, as they often appear across multiple texts without contributing significantly to the sentiment or key content.

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

Evaluation measure:

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

Methodology

Preprocessing

- **Convert all text to lowercase**

- **Special Characters**: The list of special characters was created based on a reference site and contains characters such as " !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~".

- **Stop Words**: Stop words are based on the words listed in the `stopwords.txt` file.

Sentiment Determination

According to the assignment guidelines, a 5-star rating review is considered positive, and all other ratings are considered negative.

Laplace Smoothing

Laplace smoothing has been applied under the assumption that each of the top 1000 selected words appears more than 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 among the total predictions, without differentiating between false positives, false negatives, true positives, or true negatives. This is consistent with the standard definition of accuracy: the ratio of correctly predicted instances to the total instances.

Code Details

NBC.py

```
import csv
import re

class NaiveBayesClassifier:
    def __init__(self):
        self.train_data = []
        self.test_data = []
        self.top_1000_frequent_words = []
        self.probability_of_word_positive = {}
        self.probability_of_word_negative = {}
        self.probability_of_label_positive = 0
        self.probability_of_label_negative = 0

        with open("data/stopwords.txt", "r") as file:
            stopwords = []
            for line in file.readlines():
                word = line.strip()
                stopwords.append(word)
            self.stopwords = stopwords

    # mode: "train" or "test"
    def read_data(self, filename, mode = "train"):
        with open(filename, "r", encoding="utf8") as file:
            reader = csv.reader(file)
            next(reader)  # skip header
            data = list(reader)

            if mode == "train":
                self.train_data = data
            elif mode == "test":
                self.test_data = data

    # mode: "train" or "test"
    def preprocess(self, mode = "train"):
        if mode == "train":
            data = self.train_data
        elif mode == "test":
            data = self.test_data

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

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

            # Split the text into words
```

```
            words = special_characters_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):
        # 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
        word_frequencies = list(frequent_words.items())
        word_frequencies.sort(key=lambda item: item[1], reverse=True)
        top_1000_words = [word for word, _ in word_frequencies[:1000]]
        self.top_1000_frequent_words = top_1000_words

        # Print the top 50 frequent words
        print("Top 50 Frequent Words: ")
        for i in range(50):
            print(self.top_1000_frequent_words[i])

    def train(self, percentage):
        # Select the required portion of the training data
        data_num = int(len(self.train_data) * (percentage / 100))
        data = self.train_data[:data_num]

        # Apply laplace smoothing to the word counts
        positive_word_count = [0] * len(self.top_1000_frequent_words)
        negative_word_count = [0] * len(self.top_1000_frequent_words)

        positive_reviews = 0
        negative_reviews = 0

        for review in data:
```

```
            if review[0] == 1: # If the review is positive
                positive_reviews += 1
                for word in review[1]:
                    if word in self.top_1000_frequent_words:
                        index = self.top_1000_frequent_words.index(word)
                        positive_word_count[index] += 1
            else: # If the review is negative
                negative_reviews += 1
                for word in review[1]:
                    if word in self.top_1000_frequent_words:
                        index = self.top_1000_frequent_words.index(word)
                        negative_word_count[index] += 1

        total_positive_word_count = sum(positive_word_count)
        total_negative_word_count = sum(negative_word_count)

        denominator_positive = total_positive_word_count +
len(self.top_1000_frequent_words)
        denominator_negative = total_negative_word_count +
len(self.top_1000_frequent_words)

        # Calculate the word probabilities considering Laplace smoothing
        self.probability_of_word_positive = {word:
(positive_word_count[self.top_1000_frequent_words.index(word)] + 1) /
denominator_positive for word in self.top_1000_frequent_words}
        self.probability_of_word_negative = {word:
(negative_word_count[self.top_1000_frequent_words.index(word)] + 1) /
denominator_negative for word in self.top_1000_frequent_words}

        # Calculate the class probabilities for positive and negative reviews
        self.probability_of_label_positive = positive_reviews / data_num
        self.probability_of_label_negative = negative_reviews / data_num

    def predict(self, text):
        positive_probability = self.probability_of_label_positive
        negative_probability = self.probability_of_label_negative

        for word in text:
            if word in self.probability_of_word_positive: # Check if the word is
in the positive word probabilities
                positive_probability *= self.probability_of_word_positive[word]

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

        if positive_probability > negative_probability:
            return 1  # Positive label
        else:
            return 0  # Negative label

    # Evaluate the accuracy of the Naive Bayes Classifier
```

```
    def evaluate(self):
        # Predict the labels for the test data
        predictions = []
        for review in self.test_data:
            review_text = review[1]
            prediction = self.predict(review_text)
            predictions.append(prediction)

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

        prediction_success_count = 0
        prediction_failure_count = 0

        for val1, val2 in zip(predictions, correct_answer):
            if val1 == val2:
                prediction_success_count += 1
            else:
                prediction_failure_count += 1

        accuracy = prediction_success_count / (prediction_success_count +
prediction_failure_count)

        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. The model is trained using a certain amount of training data and its accuracy is evaluated on the test dataset.

- __intit__ (self)

Initialize the NaiveBayesClassifier class by setting variables to store training and testing data, frequent words, word probabilities, and class probabilities. In addition, stop words are initialized with data read from the stopwords.txt file.

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

Reads data from the CSV file and stores it in the train_data or test_data depending on the mode provided.

- prerprocess (self, mode='train')

Preprocess the data by text transformation to lowercase, removing special characters, and stop words. The constructed data are then stored back in the train_data or test_data to ensure consistency during model training and evaluation.

- get_frequent_words (self)

Compute the frequency of each word in the training data and store the top 1000 most frequent words in the top_1000_frequent_words. It also prints the top 50 frequent words to show the most common terms in the dataset. Only 1000 words selected here will be used for training and testing.

- train (self, percentage)

Train the model using a percentage specific to the training data. It calculates the word probabilities for positive and negative reviews using Laplace smoothing and stores these probabilities in the probability_of_word_positive and probability_of_word_negative. It also calculates the overall class probabilities for positive and negative reviews and store them in probability_of_label_positive and probability_of_label_negative respectively.

- predict (self, text)

Predict the attitude of the given text by multiplying the class probability with the word probability for each word in the text. It returns the predicted label, where 1 represents a positive attitude and 0 indicates a negative attitude.

- evaluate (self)

Evaluate the performance of the Naive Bayes Classifier on the test data. It predicts the labels for the test datasets and compares them with the real labels. The model’s accuracy is then calculated and returned to measure of the model's performance.

draw.py

```python
import matplotlib.pyplot as plt

def drawPlot(dict):
    amount_of_training_data = list(dict.keys())
    accuracy = list(dict.values())

    # Draw the plot
    plt.plot(amount_of_training_data, accuracy)
    plt.scatter(amount_of_training_data, accuracy, color='r', zorder=5)

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

    # Add labels and titles
    plt.title("Model Accuracy with Varying Training Data Sizes")
    plt.xlabel("Amount of training data (%)")
    plt.ylabel("Accuracy")

    # Add accuracy values as text on the plot
    for index, (x, y) in enumerate(zip(amount_of_training_data, accuracy)):
        plt.text(x, y + 0.0003, 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 in the dictionary represent the size of the training data and the values represent the accuracy of the corresponding mode.

The drawPlot function uses the matplotlib library to generate a graph of the model's accuracy with different training data sizes. This function displays the accuracy using plt.plot and adds the circles for clear visualization using plt.scatter. The plot is labeled with a title, x-axis label, and y-axis label using plt.title, plt.xlabel, and plt.ylabel respectively. Accuracy values are added as text to the plot using repeated loops from keys and values in the input dictionary. Finally, the graph is displayed using plt.show.

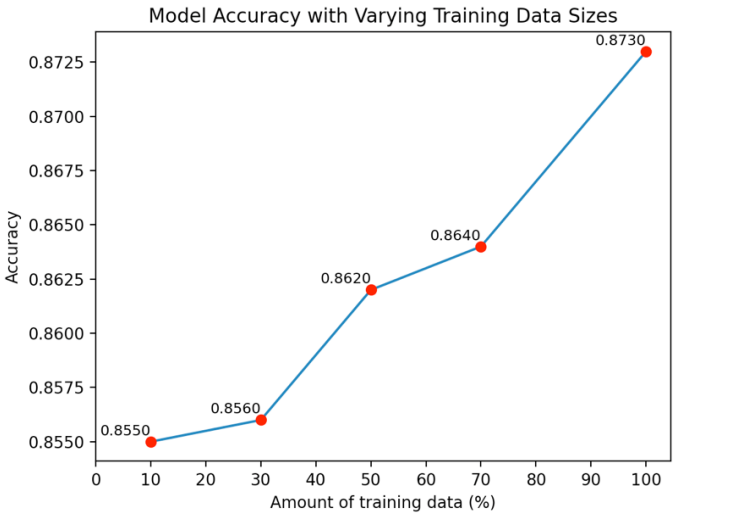
main.py

```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 code defines a NaiveBayesClassifier class, trains and tests it, and then visualizes it using the drawPlot function. The training data sizes gradually increase to 10%, 30%, 50%, 70%, and 100%, and obtain and visualize the corresponding accuracy values.

Results and Discussion

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



1. 10% Training Data

When only 10% of the training data is used, the accuracy of the model is about 0.8550, which represents the lowest accuracy observed on the graph, indicating that with a small amount of training data, the model is not very effective.

2. 30% Training Data

At 30% of the training data, the accuracy jumped slightly to about 0.8560, indicating that adding more training data helps the model learn better and improve performance.

3. 50% Training Data

As the training data grows to 50%, the accuracy increases noticeably to about 0.8620.

4. 70% Training Data

At 70% of the training data, the accuracy is about 0.8640. As in the trend so far, the accuracy has increased as the data has increased.

5. 100% Training Data

On the entire training dataset (100%), the accuracy is about 0.8730. This is the highest accuracy observed, indicating that the model performs best when all available training data are used.

The Naive Bayes classifier typically achieves between 80% and 85% accuracy on standard text classification tasks, depending on the dataset used and the preprocessing technique. This model achieves an accuracy of 87.30%, which is higher than the typical range of Naive Bayes classifiers. This suggests that the model performs very well compared to what would typically be expected from these types of classifiers.

Models show clear improvements in accuracy as more training data is provided. The most significant improvements are between 20% and 50% training data and between 70% and 100%. This analysis highlights the importance of having sufficient training data to achieve high accuracy in machine learning models. Further improvements may require more data as well as more sophisticated features or model architectures.

This model used only the conditions given in the task, but it needs additional processing about the following problems to further increase the accuracy.

- Meaningless words such as be verbs and auxiliary verbs should be excluded from learning.

- Rules are inconsistent, including positive words but excluding negative words or vice versa. Modal verbs like "can" generally carry a positive connotation, so one might hesitate to include them. And “can” is not included in stopwords.txt. However, the stopword.txt file already includes words related to "cannot." There may also be instances where "can not" is written with a space but stopwords.txt doesn’t consider about it.

- Words expressed differently by grammar are understood in different meanings in this model. For example, "want" and "wanting" are treated as different words, such as "order" and "ordered." Additionally, singular and plural forms, such as those ending in -s or -es, have the same meaning but are counted separately. Irregular forms and other grammatical differences also contribute to this problem.

- Removing special character (‘) can lead to incorrect interpretations, such as treating “i’ll” and “ill” as completely different words, and not treating them separately can distort the result. It's a problem because model judges different words as same words. In fact, I have confirmed that the words “i'll” and “ill” are used in test.csv respectively.

- Numbers themselves can't tell if they're positive or negative. That's why we need to eliminate this, too.

- Instead of analyzing single words, it is necessary to understand the context through larger units such as phrases or clauses. This is because the sentiment of individual words alone often cannot accurately determine positive or negative connotations.

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.

- Manning, Christopher, and Raghavan, Prabhakar. "Naive Bayes and Text Classification." Stanford University, https://web.stanford.edu/class/cs124/lec/naivebayes2021.pdf Accessed 5 June 2024.