Homework#2: Naive Bayes Classifier for Sentiment Analysis

Submission date: 04/08/2023
2018312164
김석진

◆Research and Study

▶ Bayes' Theorem

Probability of a hypothesis (H) given the evidence (E) is proportional to the probability of the evidence given the hypothesis, multiplied by the prior probability of the hypothesis, divided by the marginal probability of the evidence.

Mathematically: $P(H \mid E) = P(E \mid H) * P(H) / P(E)$

$P(H \mid E)$ is the posterior probability of the hypothesis given the evidence, $P(E \mid H)$ is the likelihood of the evidence given the hypothesis, $P(H)$ is the prior probability of the hypothesis, and $P(E)$ is the marginal probability of the evidence.

▶ Naïve Bayes Classifier

Probabilistic algorithm based on Bayes' Theorem. Used for both binary and multiclass classification problems. Assumes that the features are independent of each other, hence the name "naive." Simplifies calculation of conditional probabilities, making the algorithm computationally efficient and scalable.

Goal is to predict the value of the target variable given a new set of features.

- Probability of the target variable given the features: $P(Y|X') = P(X'|Y) * P(Y) / P(X')$

- Likelihood of features given the class: $P(X'|Y) = P(X1'|Y) * P(X2'|Y) * ... * P(Xn'|Y)$

- Prior probability of the class: $P(Y) = count(Y) / N$

- Evidence: $P(X') = P(X'|Y=0) * P(Y=0) + P(X'|Y=1) * P(Y=1)$

To predict the value of the target variable, calculate the probability of each class given the features and choose the class with the highest probability.

▶ Strengths:

1. Simple and fast algorithm, computationally efficient and scalable.
2. Good performance on a wide range of classification tasks, even with noisy data.
3. Robust to irrelevant features in the dataset.
4. Can achieve good performance with relatively small amounts of training data.

▶ Weaknesses:

1. Assumes independence between features, which may not hold true in some datasets.
2. Can suffer from the problem of zero probabilities.
3. Limited ability to capture complex relationships between features and target variable.
4. Sensitive to imbalanced data, where one class has significantly more instances than the other.

▶ Laplace smoothing

Laplace smoothing is a technique used to avoid zero probabilities in Naive Bayes classifier when a feature does not occur in the training data for a particular class. It adds a small positive value to each count to ensure that the probability of each feature given each class is non-zero. This helps to improve the accuracy of the model, especially for small datasets.

P(Xi|C) = (count(Xi, C) + 1) / (count(C) + k)

By using laplace smoothing we can prevent zero probability problem. And also reduce the problem of overfitting,

◆ Data preprocessing code review

```python
class PreProcessor:
    def __init__(self, stopwords_path, special_characters):
        self.stopwords_path = stopwords_path                        # stopwords file path
        self.special_characters = special_characters                # set of special characters to be removed
        self.stop_words = set()                                     # an empty set to hold the stop words

    def load_stopwords(self):
        with open(self.stopwords_path, 'r') as f:                   # open the stopwords file in read mode
            self.stop_words.update([word.strip() for word in f])    # add the stop words to the set

    def preprocess_text(self, text):
        # remove special characters
        clean_text = ''
        for char in text:
            if char not in self.special_characters:
                clean_text += char

        # split into words
        words = clean_text.split()

        # remove stop words
        words = [w for w in words if w not in self.stop_words]
        return words
```

```python
def process_file(self, file_path):
    self.load_stopwords()                                   # load the stop words

    # preprocess the text and count word frequencies
    word_counts = Counter()                                 # create a counter to count word frequencies

    # open the input file
    with open(file_path) as f:
        reader = csv.DictReader(f)                          # create a dictionary reader object

        # iterate over each row in the file
        for row in reader:
            text = row['text'].lower()                      # get the text in the current row and convert to lowercase
            words = self.preprocess_text(text)              # preprocess the text in the current row
            word_counts.update(words)                       # update the counter with the preprocessed words

    # select the top 1000 words by frequency as word features
    word_features = [w for w, _ in word_counts.most_common(1000)]       # get the 1000 most common words from the counter
    return word_features                                                # return the selected word features

def top_20_words(self, word_features):
    print("\n<<<Top 20 words>>>")
    print(word_features[:20])                   # print the top 20 words
```

<code explanation>

I made a PreProcessor class to group all the thing I need to preprocess the data set.

Class Variables:

1. stopwords_path: the file path of the stopwords file
2. special_characters: a set of special characters to be removed

Methods:

1. load_stopwords(self): Method to load the stopwords from the provided file path and add them to the stop_words set.

2. preprocess_text(self, text): Method to preprocess a given text by removing special characters, tokenizing it into words, and removing stop words. Returns a list of preprocessed words.

   This method works in these steps

   First removes special characters from the text, then tokenizes the text into words. Removes stop words from the words and returns a list of preprocessed words

3. process_file(self, file_path): Method to preprocess the text in the file at the given file path, count the frequency of each word, and select the top 1000 most frequent words as word features. Returns a list of the selected word features.

   This method works in these steps

   Loads the stopwords from the provided stopwords file path then iterates over each row in the file at the given file path.

   Gets the text in the current row and preprocesses it using the preprocess_text method. Counts the frequency of each word in the preprocessed text using a Counter.

   And finally elects the top 1000 most frequent words as word features.

4. top_20_words(self, word_features): Method to print the top 20 words in the provided list of word features.

◆ Training and Prediction code review

```python
class NaiveBayesClassifier:
    def __init__(self, features, myPerProcess):
        # Initialize the class with a list of features and an instance of the pre-processing class
        self.features = features
        self.word_probs = []  # This will store the probabilities of each word in each class
        self.train_file_path = 'train.csv'  # File path to the training data
        self.myPerProcess = myPerProcess  # Instance of the pre-processing class

    def train(self, train_data_ratio, k):
        print("\n<<<Training>>>")
        print("Loading stopwords...")
        self.myPerProcess.load_stopwords()  # Load the stop words for pre-processing

        # in this part the we count the 5-star and 1-star review in all the data
        print("Counting positive and negative instances in training set...")
        # variable to count the 5 star, 1 star rating in every features
        pos_count = 0
        neg_count = 0
        with open(self.train_file_path) as f:
            reader = csv.DictReader(f)

            # Read in the rows from the training data
            rows = [row for row in reader]
            num_rows = len(rows)
            train_rows = rows[:int(train_data_ratio*num_rows)]
            for row in train_rows:
                # Count the positive instances (5-star reviews)
                if row['stars'] == '5':
                    pos_count += 1
```

```python
                # Count the negative instances (1 star reviews)
                else:
                    neg_count += 1

        # in this part the we count the occurrences of each word in 5-star and 1-star review
        print("Counting occurrences of each feature in positive and negative instances...")
        pos_word_count = Counter()
        neg_word_count = Counter()
        for row in train_rows:
            # Count the occurrences of each word in positive instances
            if row['stars'] == '5':
                pos_word_count.update(self.myPerProcess.preprocess_text(row['text']))
            # Count the occurrences of each word in negative instances
            else:
                neg_word_count.update(self.myPerProcess.preprocess_text(row['text']))

        # in this part the calculate 5-star and 1-star probability in every feature
        print("Computing probabilities of each feature given the target class...")
        for word in self.features:
            # Compute the probability of each word given the target class using Laplace smoothing
            pos_word_prob = (pos_word_count[word] + 1) / (pos_count + k)
            neg_word_prob = (neg_word_count[word] + 1) / (neg_count + k)

            # Add the word probabilities to the list
            self.word_probs.append((word, pos_word_prob, neg_word_prob))
```

```python
def predict(self, test_file_path):
    print("\n<<<Predicting>>>")
    # Load stopwords
    self.myPerProcess.load_stopwords()

    # Preprocess the test data and store it in a list
    preprocessed_texts = []
    with open(test_file_path) as f:
        reader = csv.DictReader(f)
        for row in reader:
            preprocessed_texts.append(self.myPerProcess.preprocess_text(row['text']))

    # Initialize true positive, false positive, true negative, and false negative counters
    true_pos = 0
    false_pos = 0
    true_neg = 0
    false_neg = 0

    # Open the test file and iterate over each row
    with open(test_file_path) as f:
        reader = csv.DictReader(f)
        counter = 0
        for i, row in enumerate(reader):
            # Initialize the probability ratios for positive and negative classes
            pos_ratio = 1
            neg_ratio = 1
            # Iterate over each word in the preprocessed text and update the probability ratios
            for word, pos_word_prob, neg_word_prob in self.word_probs:
                if word in preprocessed_texts[i]:
                    pos_ratio *= pos_word_prob
                    neg_ratio *= neg_word_prob
```

```python
                else:
                    pos_ratio *= 1 - pos_word_prob
                    neg_ratio *= 1 - neg_word_prob
            # Compute the probability of the review being positive and negative using the computed probability ratios
            # To avoid division by zero error, a very small value (1e-10) is added to the denominator
            pos_prob = pos_ratio / (pos_ratio + neg_ratio + 1e-10)
            neg_prob = neg_ratio / (pos_ratio + neg_ratio + 1e-10)
            # Make the final prediction based on the probability
            if pos_prob > neg_prob:
                prediction = '5'
            else:
                prediction = '1'
            # Update the true positive, false positive, true negative, and false negative counters based on the prediction
            if prediction == row['stars']:
                if prediction == '5':
                    true_pos += 1
                else:
                    true_neg += 1
            else:
                if prediction == '5':
                    false_pos += 1
                else:
                    false_neg += 1

            # Print the progress of prediction for every 250th review
            counter += 1
            if counter % 250 == 0:
                print(f"Processed {int(counter/10)}% of test data...")
```

```python
# Compute accuracy, precision, recall, and f1 score based on
# the true positive, false positive, true negative, and false negative counters
accuracy = (true_pos + true_neg) / (true_pos + true_neg + false_pos + false_neg)
precision = true_pos / (true_pos + false_pos)
recall = true_pos / (true_pos + false_neg)
f1 = 2 * precision * recall / (precision + recall)


print("\n<<<Prediction Results>>>")
print("==================================================")
print(f"|              |  Actual Positive |  Actual Negative |")
print(f"|-------------|-----------------|-----------------|")
print(f"|Predicted Pos. |        {true_pos:3d}         |        {false_pos:3d}        |")
print(f"|Predicted Neg. |        {false_neg:3d}         |        {true_neg:3d}        |")
print("--------------------------------------------------")
print(f"Accuracy:  {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall:    {recall:.2f}")
print(f"F1 Score:  {f1:.2f}")

return accuracy
```

<code explanation>

I made an NaiveBayesClassifier class to group all the variable and method to train and predict using Naïve Bayes method.

Class Variables:

1. features: a list of features to be used in training the classifier

2. word_probs: a list to store the probabilities of each word in each class

3. train_file_path: a string variable that holds the file path to the training data

4. myPerProcess: an instance of the pre-processing class

Method:

1. train(self, train_data_ratio, k)

   This method trains the Naive Bayes Classifier on the training data. The train_data_ratio parameter is used to split the training data into a training set and a validation set. The k parameter is used in Laplace smoothing to avoid zero probabilities.

   a. Loads stop words for pre-processing. And counts the number of positive and negative instances in the training data.

   b. Counts the occurrences of each feature in positive and negative instances.

c. Computes the probabilities of each feature given the target class (5-star or 1-star) using Laplace smoothing.

By step a. b. c. we calulate the prior and likelihood.

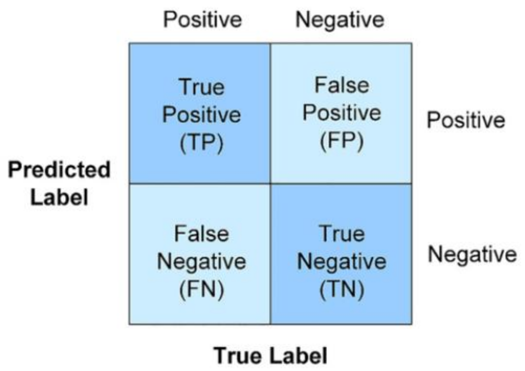
2. predict(self, test_file_path):

This code uses the probability calculate by train() method to predict the rating(5-star, 1-star) in the text data set and comparing the real rating.

a. For each row, it initializes the probability ratios for positive and negative classes to 1 and iterates over each word in the preprocessed text to update the probability ratios based on the occurrence of each word in the training data.

b. It then computes the probability of the review being positive and negative using the computed probability ratios and makes the final prediction based on the probability. The code updates the true positive, false positive, true negative, and false negative counters based on the prediction.

By these two steps the prediction of the review is made. And it compares the result with real value.



$$Accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}$$

$$Precision = \frac{T_p}{T_p + F_p}$$

$$Recall = \frac{T_p}{T_p + T_n}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

It compares and computes the accuracy, precision, recall, and f1 score based on the true positive, false positive, true negative, and false negative counters.

◆ Visualization code review

```python
class Optimize:
    def __init__(self, features, myPerProcess):
        self.features = features
        self.myPerProcess = myPerProcess


    def plot_learning_curve(self):
        training_sizes = [0.1, 0.3, 0.5, 0.7, 1] # fraction of total dataset
        accuracy_scores = []
        k = 1024

        for ratio in training_sizes:
            print(f"\n<<<{int(ratio * 100)}% data>>>")

            classifier = NaiveBayesClassifier(self.features, self.myPerProcess)

            # Train the classifier on the specified number of training samples
            classifier.train(ratio, k)

            # Test the classifier on the test dataset and record the accuracy score
            score = classifier.predict('test.csv')
            accuracy_scores.append(score)

        # Find the index of the maximum accuracy score and use it to get the corresponding training size
        max_score = max(accuracy_scores)
        max_index = accuracy_scores.index(max_score)
        best_training_size = training_sizes[max_index]
```

```python
        # Plot the learning curve
        plt.plot(training_sizes, accuracy_scores, '-o')
        plt.xlabel('Training Set Size')
        plt.ylabel('Accuracy Score')
        plt.title('Learning Curve Analysis')
        plt.show()

        return best_training_size
```

```python
    def plot_laplace_curve(self, best_training_size):

        laplace_param = [1, 4, 16, 64, 256, 1024, 4096]
        accuracy_scores = []

        for k in laplace_param:
            print(f"\n<<<k = {k}>>>")
            classifier = NaiveBayesClassifier(self.features, self.myPerProcess)

            # Train the classifier on the specified laplace_param
            classifier.train(best_training_size, k)

            # Test the classifier on the test dataset and record the accuracy score
            score = classifier.predict('test.csv')
            accuracy_scores.append(score)

        # Plot the laplace curve
        plt.plot(laplace_param, accuracy_scores, '-o')
        plt.xlabel('Laplace Parameter')
        plt.ylabel('Accuracy Score')
        plt.title('Laplace Curve Analysis')
        plt.show()
```

<code explanation>

I made an Optimize class to group all the variable and method to visualize the data and optimize the training set

Class Variables:

1. features: the selected 1000 most frequent words by PreProcessing class

2. myPerProcess: an instance of the PreProcessor class for text preprocessing

Methods:

1. plot_learning_curve(self):

   This method shows a learning curve with variety of data set size.

   And returns the best suitable data size.

2. plot_laplace_curve(self, best_training_size)

   Get the value from plot_learning_curve and show a Laplace curve with variety of laplace parameters.

◆ Result

Main.py: for execution I made a main.py file to use PreProcessor class, NaïveBayesClassifier class, and Optimize class

<TASK 1>

```python
from preprocessing import PreProcessor
from training import NaiveBayesClassifier
from analysis import Optimize
import matplotlib.pyplot as plt


stopwords_path = 'stopwords.txt'
special_characters = set(['!', '\"', '#', '$', '%', '&', '\'', '(', ')', '*', '+', ',', '-', '.', '/',
                          ':', ';', '<', '=', '>', '?', '@', '[', '\\', ']', '^', '_', '`', '{', '|', '}', '~'])

"""-----------Task 1: Feature Selection-----------------"""
print("=====================================Task 1: Feature Selection=====================================")

# make a preprocess object to preprocess the taining data using stapword file and special characters as input
myPerProcess = PreProcessor(stopwords_path, special_characters)           # make a preprocess object
features = myPerProcess.process_file('train.csv')                         # save a 1000 most recently appeared words as features

# print the 20 most recently appeared words
myPerProcess.top_20_words(features)

input("Press Enter to continue...")
```

<TASK 2>

```python
"""-----------Task 2: Model Training and Evaluation-----------------"""
print("\n\n=====================================Task 2: Model Training and Evaluation=====================================")

# NaiveBayesClassifier class contains a training code and prediction code
# input: feature(1000 words), PreProcessor object
myClassifier = NaiveBayesClassifier(features, myPerProcess)             # make a NaiveBayesClassifier object

# for task 2 I randomly choose the data set size to be 100% and laplace smoothing parameter to be 3
# at task 3 I will optimize using various value
data_ratio = 1
k = 2000

# train the model using train() method
myClassifier.train(data_ratio, k)

# using the model trained evaluate the performance using predict() method
accuracy = myClassifier.predict('test.csv')


input("Press Enter to continue...")

```

<TASK 3>

```python
"""-----------Task 3: Learning Curve Analysis-----------------"""
print("\n\n=====================================Task 3: Learning Curve Analysis=====================================")

# Optimize class contains a method to optimize the model and other funtions to visialize and evaluate the performance
optimize = Optimize(features, myPerProcess)

# at plot_learning_curve funtion I try various data set size and returns the size with best accuracy
best_training_size = optimize.plot_learning_curve()

# at plot_laplace_curve funtion I try various laplace parameter and plot the performance
optimize.plot_laplace_curve(best_training_size)

input("Press Enter to exit...")
```

Task 1 Result

In Task 1 we need to tokenize all the review and get 1000 most frequently used words in the train data to use these words as a feature. And print 20 most frequently from 1000 words.

```
==================================Task 1: Feature Selection==================================

<<<Top 20 words>>>
['food', 'place', 'just', 'like', 'here', 'time', 'back', 'great', 'service', 'will', 'can', 'only', 'dont', 'im',
'ive', 'went', 'know', 'people', 'well', 'best']
Press Enter to continue...
```

<result analysis>

We can see that food is the most recently used word in the review. And 20 words are represented in the terminal.

Task 2 Result

In Task 2 we need to train the model using train data and using test data predict the rating (5-star, 1-star) and compare the result with real rating.

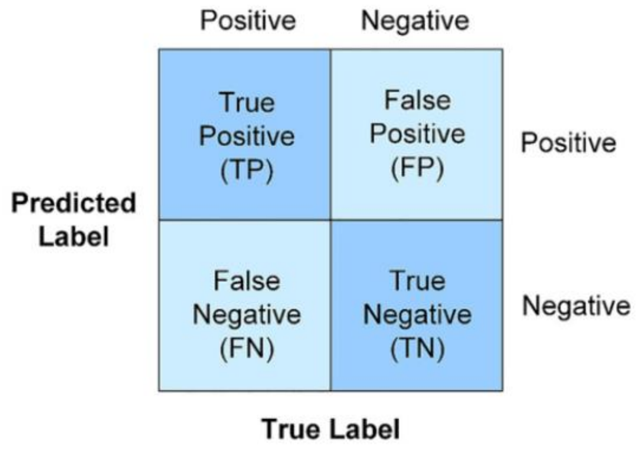
```
=====================================Task 2: Model Training and Evaluation=====================================

<<<Training>>>
Loading stopwords...
Processed 50% of test data...
Processed 75% of test data...
Processed 100% of test data...

<<<Prediction Results>>>
==================================================
|              |  Actual Positive |  Actual Negative |
|--------------|------------------|------------------|
|Predicted Pos. |      399        |        95        |
|Predicted Neg. |      131        |       375        |
--------------------------------------------------
Accuracy:  0.77
Precision: 0.81
Recall:    0.75
F1 Score:  0.78
Press Enter to continue...
```

<result analysis>

I tried to use multiple value to evaluate the performance of the model. We can see in the terminal that the Accuracy is 77%, Precision is 81%, Recall is 75%, and F1 score is 78%. As we can see that all the values are having a high value so the performance is rather good and evenly good.



$$Accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}$$

$$Precision = \frac{T_p}{T_p + F_p}$$

$$Recall = \frac{T_p}{T_p + T_n}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

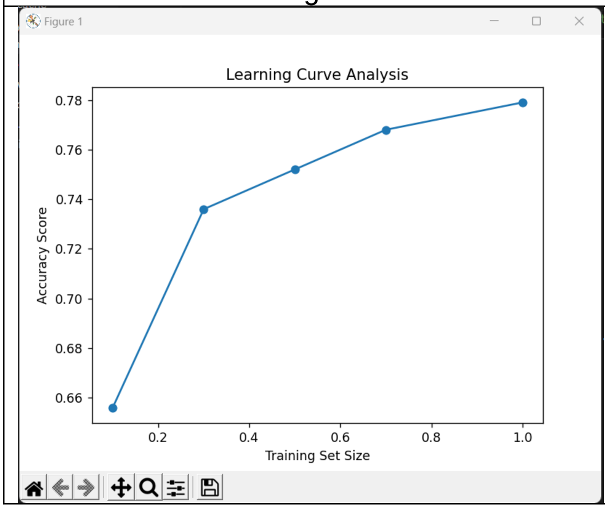
For a little error analysis we can see that FN(false negative) value is rather high so one might think that the model is predicting 1-star better. But if we try to calculate the ratio, 131/(399+131) and 95/(375+95) are similar so the performance is evenly good.


```
==================================================
|              |  Actual Positive |  Actual Negative |
|--------------|------------------|------------------|
|Predicted Pos. |      399        |        95        |
|Predicted Neg. |      131        |       375        |
--------------------------------------------------
```

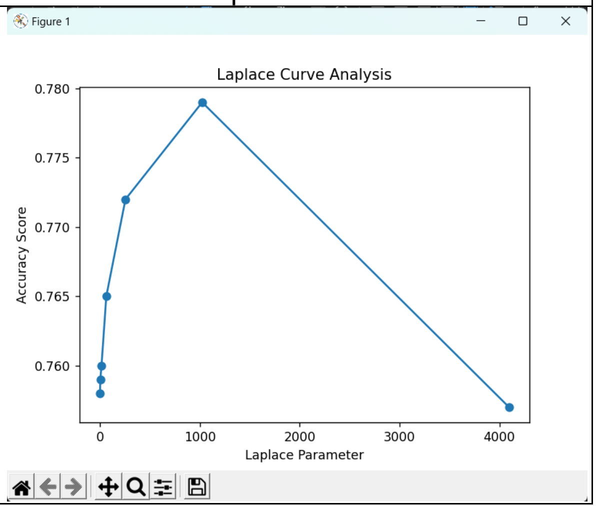
Task 3 Result

In task 3 I tried to visualize and optimize the Naïve Bayes Model. So I tries to plot two graph. First graph is a learning curve that shows the accuracy with change of train data size. Second graph shows the accuracy with change of laplace parameter.

<Learning Curve>

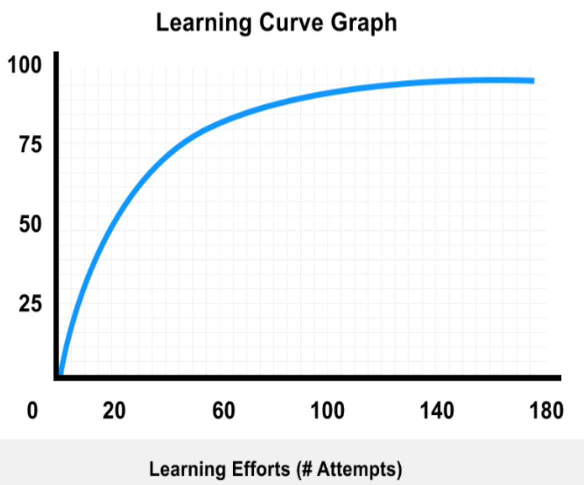


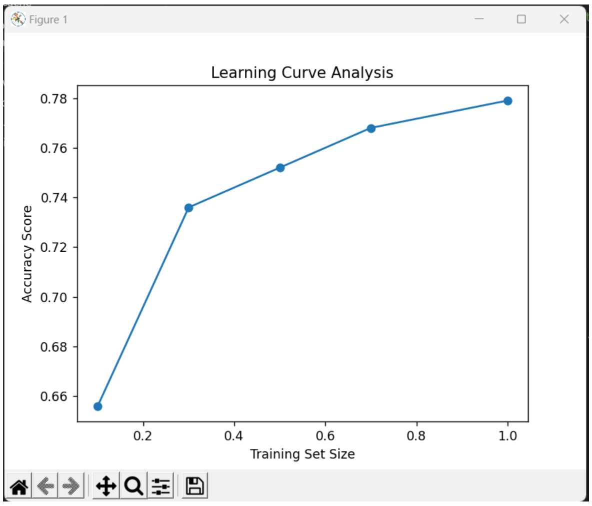
<Laplace Curve>



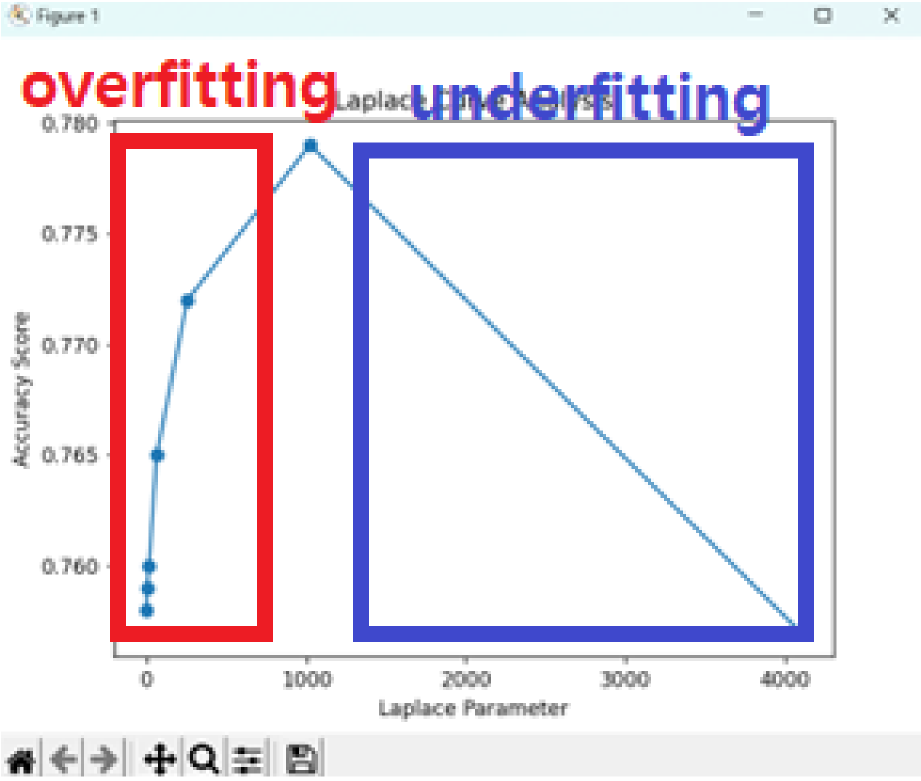
<result analysis>

Every model have a different Learning Curve so there is no perfect form of a learning curve, But as the training size increase accuracy is recommend to incline as well. And comparing the grapth with ideal learning curve we can see that the learning curve is well shaped





And for the Laplace Curve, as the purpose of Laplace smoothing is solving the zero prob problem and improving overfitting. So there must a certain laplace parameter that will give the best performance, just between overfitting and underfitting.



As in the picture above the model was highly overfitted when k = 1, and as k increases the performance increases and peaks at k=1024 and after that gets too underfit. So using value k = 1024 will gives the best performance 78%

◆ Optimization

Code above is actually after the attempt of optimization. So I will elaborate the attempts to increase the performance of the model

Method 1 to optimize spead: Counter

To increase the spead of searching and training I used Counter class in collection package. Before using Counter I use for loop to preprocess training data and training. And it took tons of times.

```
for word in words:
    if word not in word_counts:
        word_counts[word] = 1
    else:
        word_counts[word] += 1
```

So I changed the code like this to use the Counter class. By updating the counter object it fastly count the value I need.

```
# preprocess the text and count word frequencies
word_counts = Counter()                                              #
create a counter to count word frequencies

# open the input file
with open(file_path) as f:
    reader = csv.DictReader(f)                                       #
    create a dictionary reader object

    # iterate over each row in the file
    for row in reader:
        text = row['text'].lower()                                   #
        get the text in the current row and convert to lowercas
        words = self.preprocess_text(text)                           #
        preprocess the text in the current row
        word_counts.update(words)                                    #
        update the counter with the preprocessed words
```

Method 2 to optimize spead: Divide the code in classes

I tried to optimize the code to prevent the model doing iterative code as less as possible so I divide the classes in three and save the unchanging reused value as much to increase speed.

```
class PreProcessor:
    def __init__(self, stopwords_path, special_characters):
        self.stopwords_path = stopwords_path                          #
        stopwords file path
        self.special_characters = special_characters                  #
        set of special characters to be removed
        self.stop_words = set()                                       # an
        empty set to hold the stop words
```

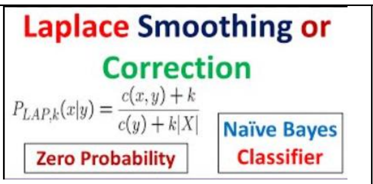
```
class NaiveBayesClassifier:
    def __init__(self, features, myPerProcess):
        # Initialize the class wi[illegible] features and an instance
        of the pre-processing class
        self.features = features
        self.word_probs = []  # This will store the probabilities of
        each word in each class
        self.train_file_path = 'train.csv'  # File path to the training
        data
        self.myPerProcess = myPerProcess  # Instance of the
        pre-processing class
```

```
class Optimize:
    def __init__(self, features, myPerProcess):
        self.features = features
        self.myPerProcess = myPerProcess
```

For example as we can see when making a classifier object or optimize class PreProcessor class is passed. That is because PreProcessing is needed in both class but the preprocessing is only need for one training data. So I made only one PreProcessing object and passed these to other objects.

Method 3 to increase performance: Using different Laplace Smoothing method and Laplace parameter

As I was searching for performance improvement I found that there are multiple ways to apply Laplace Smoothing. So I decided to try different variation.



$$\frac{count(w_i,c)+1}{\left(\sum_{w\in V} count(w,c)\right) + |V|}$$

$$P(X|Y=k) = \frac{T_{N,k}}{T_k} \rightarrow \frac{T_{N,k}+\alpha}{T_k+2\alpha}$$

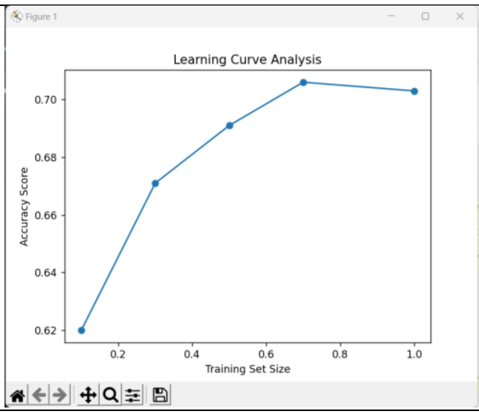
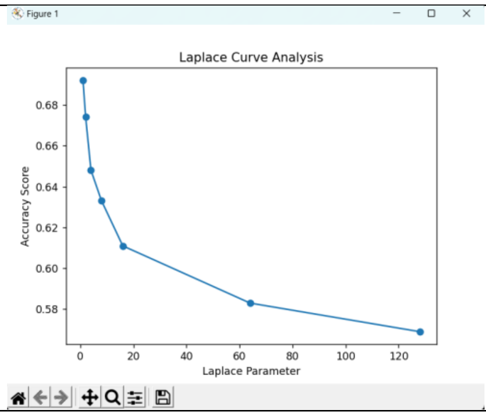
With a different Laplace smoothing method there were different learning curve and different Laplace curve and I tried different value, different variation. I do not have all the result but I have one with second method.

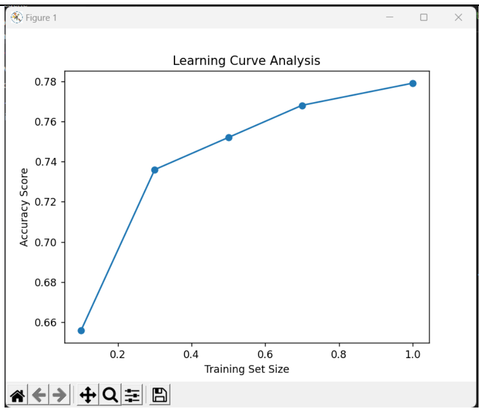
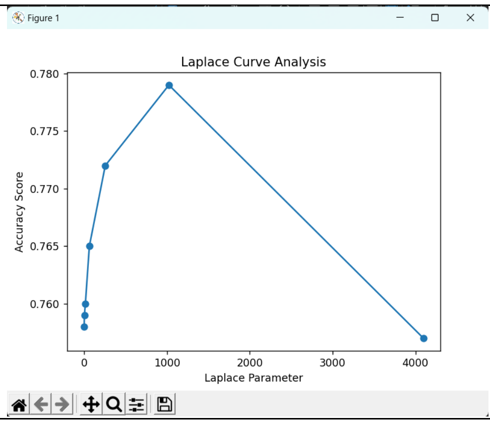
```
pos_word_prob = (pos_word_count + k) / (pos_count + (2 * k))
neg_word_prob = (neg_word_count + k)/ (neg_count + (2 * k))


pos_word_prob = (pos_word_count + k) / (pos_count + (k * train_data_size))
neg_word_prob = (neg_word_count + k) / (neg_count + (k * train_data_size))


pos_word_prob = (pos_word_count + 1) / (pos_count + k)
neg_word_prob = (neg_word_count + 1) / (neg_count + k)
```

<Performance increase>

<table>
<tr><td></td><td><Learning Curve></td><td><Laplace Curve></td></tr>
<tr><td><method 2></td><td>
</td><td>
</td></tr>
</table>

<method 3>		

Just a brief analysis with method 2 we can see that with only 1000 train data size the performance is decreasing slightly so it might be getting overfitted. But in method 3 the accuracy is still increasing as the data size is increasing. So se can expect that with a more larger data set the performance of method 3 will be better.

And for the laplace curve changing the parameter in method 2 is having no improvement as k gets bigger the accuracy is decreasing exponencially. But in method 3 we can find a best value k with highest performance.

With this analysis I found the optimized code and parameter is method 3 with k value between 1000~1200, and a larger train data set size.