Name: Shrutika Shrutika USC ID: 7908890187

Q1) Section: Tokenizer

The Tokenizer class is responsible for: Tokenizing text into individual words, Removing stop words, Building a vocabulary based on word frequency from the training texts, Converting texts into binary feature vectors where each word is represented as 1 or 0.

```
        text = text.lower()
            # Remove punctuations
        tokens = re.findall(r'\b\w+\b', text)
        # Remove stop words
        tokens = [token for token in tokens if token not in self.stop_words]
        return tokens
```

My Reasoning:

Lowercasing: I have changed the text to lowercase in order to standardize the token set. Ex: "good" and "Good" will be treated the same.

Removing Punctuation: This step will remove Punctuation since it does not carry sentiment and this helps in eliminating unnecessary terminology. As our task is a sentiment classification problem, which requires the overall sentiment, that is, positive/negative, therefore, I believe punctuation such as comma, full stop, and question mark are of no direct use in getting the sentiment of the reviews.

Removing Common Stop Words: In order for words relevant to sentiment to be focused on, words such as "the" and "is" are removed and some important stop words like not, nor etc are not removed since they add value for positive or negative sentiment.

Tokenization: The regex r'\b\w+\b' will extract the words, ensuring we capture only meaningful tokens without punctuation.

Vocabulary Size: The top 10,000 most frequently occurring terms is used as a trade-off between computational efficiency and relevancy.

Texts are represented as binary vectors 'words present or not', that simplifies the model, putting emphasis on the presence of words related to sentiments.

Q2) Section: Results

A/c to class slides i have used the following formulas for Evaluation Metrics:

- Accuracy: Measures the proportion of correct predictions out of all predictions (TP + TN) / (TP + FN + FP + TN)
- Precision: Represents how many of the predicted positives are actual positives TP / (TP + FP)
- Recall: Reflects how many actual positives were correctly predicted. TP / (TP + FN)
- F-Measure: It is a balance between precision and recall, the harmonic mean of the two 2 * (Precision * Recall) / (Precision + Recall)

```
 # Taking this formula from slides
accuracy=(true_positives+true_negatives)/ (true_positives+false_negatives+false_positives+true_negatives)
precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
f1 = 2 * (precision * recall) / (precision + recall)
```

Once these metrics have been applied to the model's predictions in val_predictions.csv against the true labels in val_labels.csv, the following results are obtained:

	K=1000 (with stop word removal)	K=10000 (without stop word removal)	**K=10000 (with stop word removal)**
Val Accuracy	0.8350	0.8534	**0.8570 (85.70%)**
Val Precision	0.8222	0.8558	**0.8653 (86.53%)**
Val Recall	0.8547	0.8499	**0.8455 (84.55%)**
Val F1-Score	0.8381	0.8528	**0.8553 (85.53%)**

It means that this model has achieved a good balance between precision and recall, which is reflected in a high F1-score for effectiveness in correctly choosing both positive and negative reviews. An overall accuracy of 85.70% states that the model performs well with respect to general correctness of prediction.

Q3) Section: Naive Bayes

Training Procedure

Initialization: It initializes the NaiveBayesClassifier with parameters like alpha, which generally is a smoothing factor to avoid zero probabilities. This controls how unseen words are handled during inference.

```
class NaiveBayesClassifier:
   def __init__(self, alpha=1.0):
```

Calculating Prior Probabilities: It calculates the prior probabilities for the positive and negative classes of sentiments by counting, in each class, how many reviews belong to that class during the training process.

```
       self.prior_positive = num_positive / num_docs
       self.prior_negative = num_negative / num_docs
```

Count Word Occurrences: Then, it goes through each word in the vocabulary and counts how many times it appeared in a positive and in a negative review, storing that count. For each word, the model sums up how many times it appears in positive reviews and how many times it appears in negative reviews, storing this in `positive_counts` and `negative_counts`, respectively.

```
       # Count word occurrences in positive and negative documents
       positive_counts = np.zeros(X.shape[1])
       negative_counts = np.zeros(X.shape[1])
       for i, label in enumerate(y):
           if label == 1:
               positive_counts += X[i]
           else:
               negative_counts += X[i]
```

Computing Conditional Probabilities: Using Laplace Smoothing
For words not in each review, Laplace smoothing adds a small value to each count so that all the words become non-zero. The counts then are turned into conditional probabilities of the word given it's in a positive or negative review.

positive_word_probs: The probability of a word appearing in a positive review.
negative_word_probs: The probability of a word appearing in a negative review. These probabilities are adjusted using the smoothing factor alpha to ensure no probability is zero.

```
# Apply Laplace smoothing and calculate conditional probabilities # Smoothing factor alpha
self.positive_word_probs=(positive_counts+self.alpha) / (num_positive + self.alpha * self.vocab_size)
self.negative_word_probs=(negative_counts + self.alpha)/ (num_negative + self.alpha * self.vocab_size)
```

Here, self.alpha is the smoothing factor.

Inference:

Calculating Log-Probabilities : This model will, during prediction, for each review, compute the log-probability of it being positive or negative by summing the logs of word probabilities for all words in the review learned during training.

```
   def predict(self, X):
       # Calculate log-probabilities for each class
       log_prior_positive = np.log(self.prior_positive)
       log_prior_negative = np.log(self.prior_negative)

       positive_scores = X @ np.log(self.positive_word_probs) + (1 - X) @ np.log(1 - self.positive_word_probs)
       negative_scores = X @ np.log(self.negative_word_probs) + (1 - X) @ np.log(1 - self.negative_word_probs)
```

This code calculates the log-probabilities for each class by multiplying the binary feature vectors (X) with the log of the word probabilities and summing up the results.

Scoring:

The model calculates the **positive score** as the sum of the log-probabilities of each word being in a positive review.
Similarly, the **negative score** is calculated by adding the log-probabilities for the negative class. (Described in above code)

Prediction: The model compares the final scores (positive and negative). If the positive score is greater, it predicts the document as positive; else, it predicts the document as negative.

```
return (log_prior_positive + positive_scores) >= (log_prior_negative + negative_scores)
```