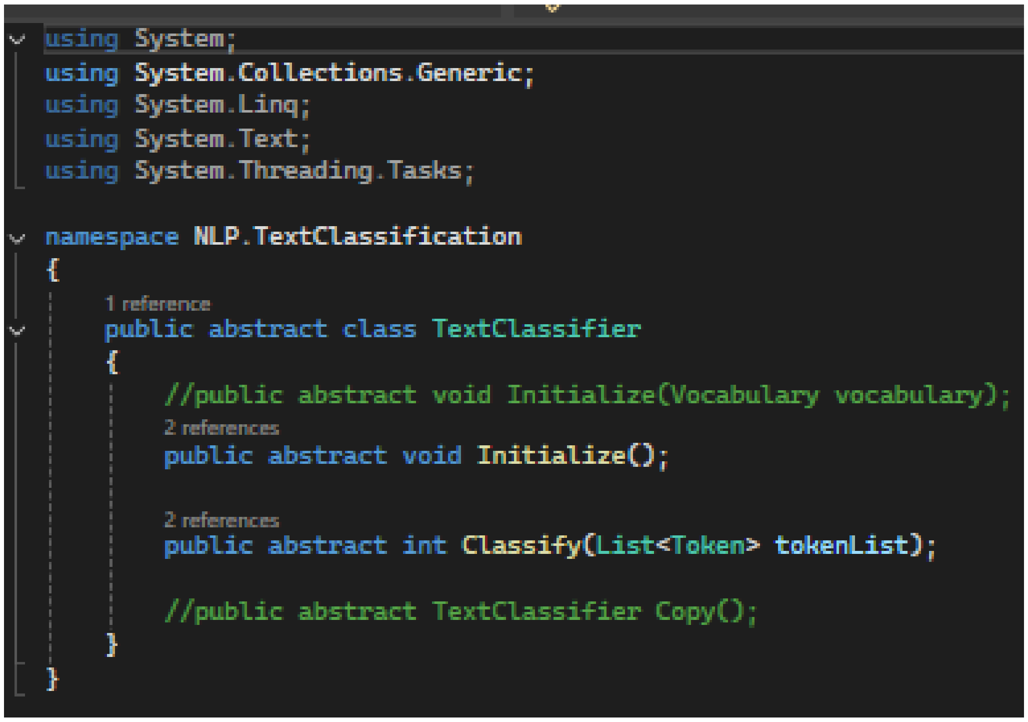
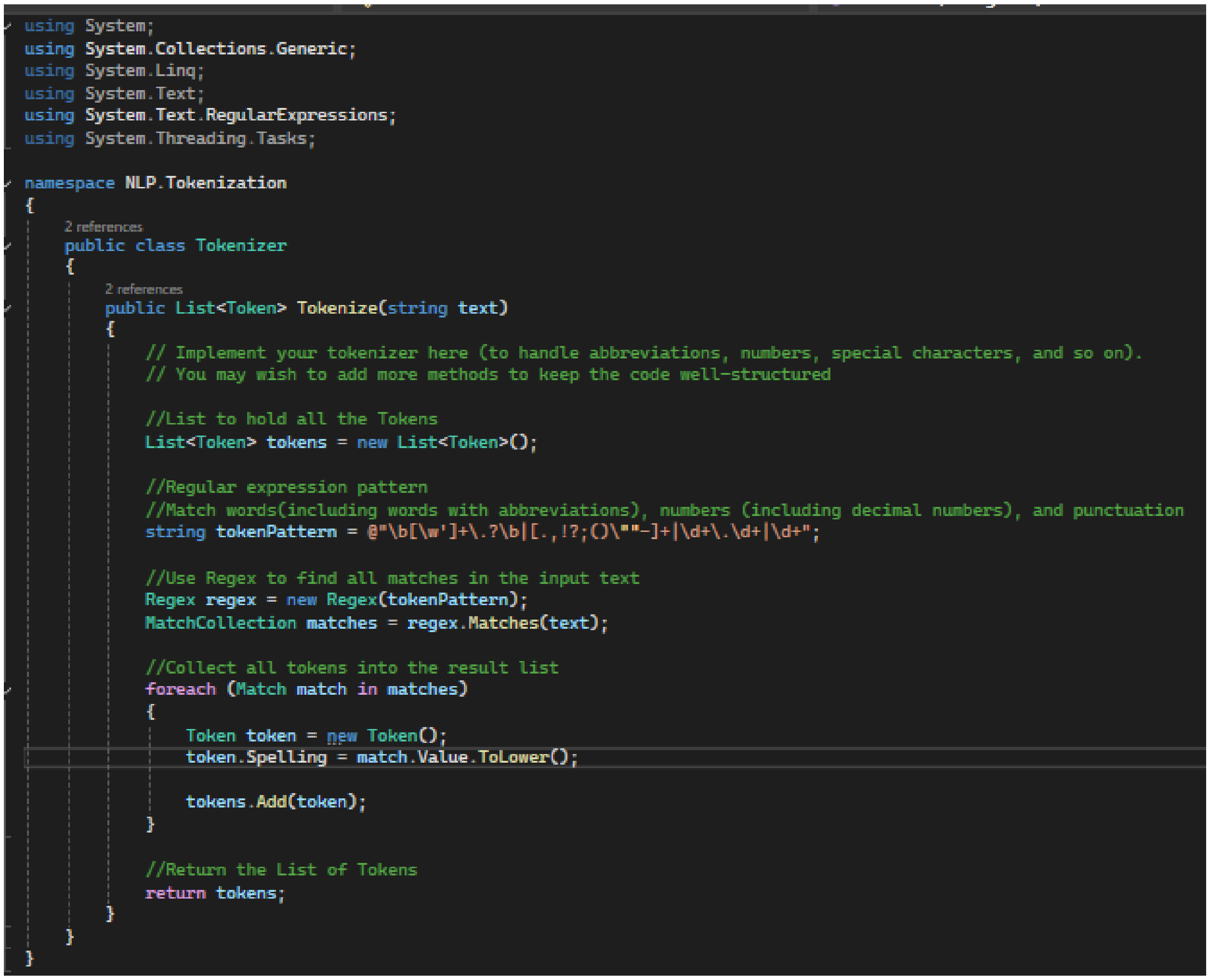
Explanation of C# implementation



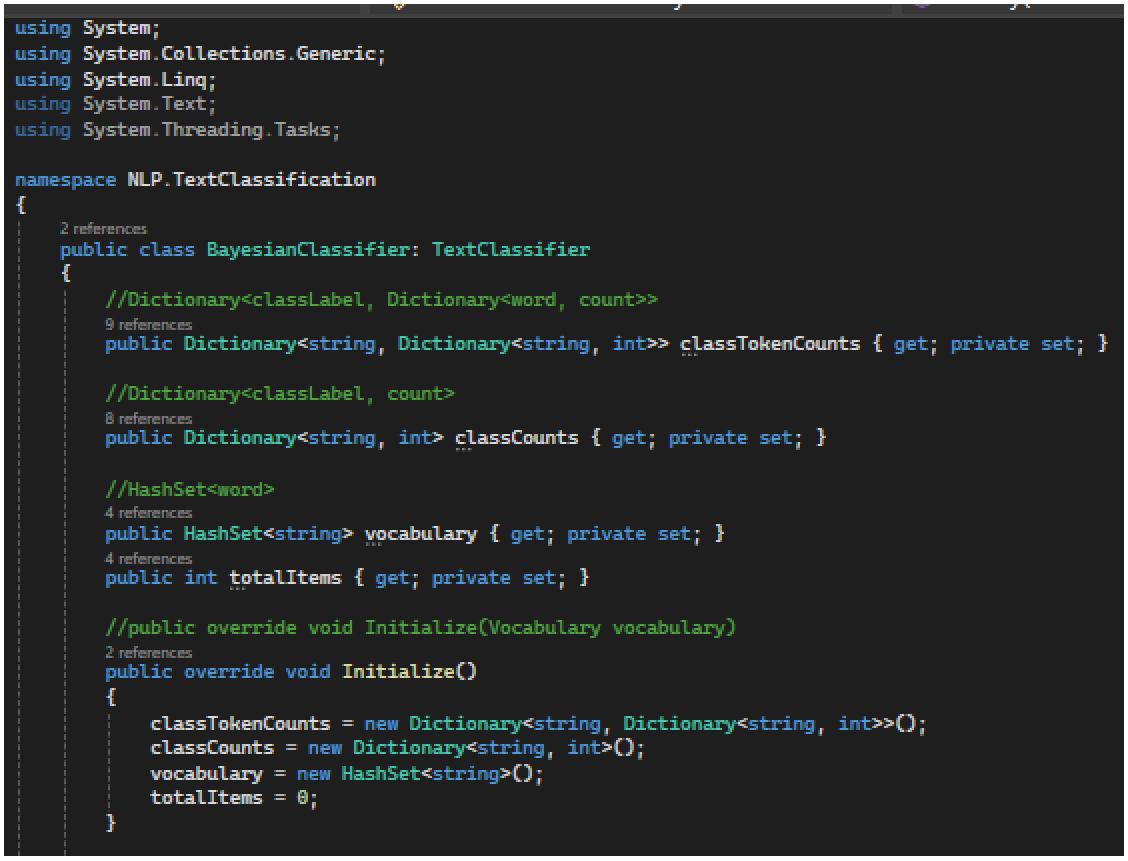
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace NLP.TextClassification
{
    1 reference
    public abstract class TextClassifier
    {
        //public abstract void Initialize(Vocabulary vocabulary);
        2 references
        public abstract void Initialize();

        2 references
        public abstract int Classify(List<Token> tokenList);

        //public abstract TextClassifier Copy();
    }
}
```

In the figure above, TextClassifier has the abstract methods, void Initialize() and int Classify(List<Token> tokenList). The Copy() method will not be used and is commented out. The Vocabulary class, which was used as a parameter in the Initialize() method in other parts of the assignment, will not be used here and is also removed.



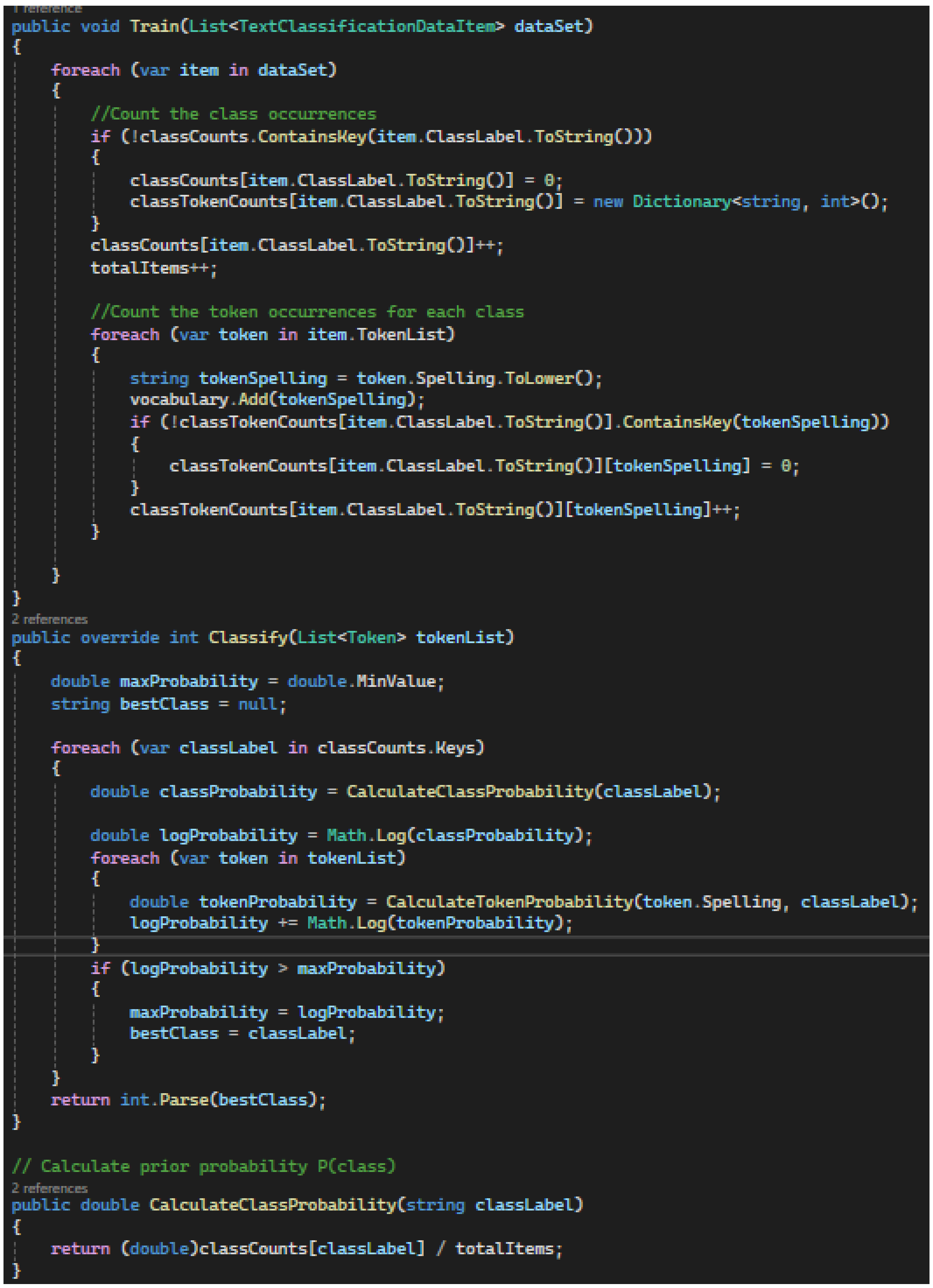
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace NLP.Tokenization
{
    2 references
    public class Tokenizer
    {
        2 references
        public List<Token> Tokenize(string text)
        {
            // Implement your tokenizer here (to handle abbreviations, numbers, special characters, and so on).
            // You may wish to add more methods to keep the code well-structured

            //List to hold all the Tokens
            List<Token> tokens = new List<Token>();

            //Regular expression pattern
            //Match words(including words with abbreviations), numbers (including decimal numbers), and punctuation
            string tokenPattern = @"\b[\w']+\.?\b|[.,!?;()\""-]+|\d+\.\d+|\d+";

            //Use Regex to find all matches in the input text
            Regex regex = new Regex(tokenPattern);
            MatchCollection matches = regex.Matches(text);

            //Collect all tokens into the result list
            foreach (Match match in matches)
            {
                Token token = new Token();
                token.Spelling = match.Value.ToLower();

                tokens.Add(token);
            }

            //Return the List of Tokens
            return tokens;
        }
    }
}
```

In the figure above, the Tokenizer class has the Tokenize method return a List of Tokens and has string text as its parameter. Using Regular Expressions, words with abbreviations, numbers and punctuations are extracted from the text, and then added to a Token. A List of these Tokens is formed and returned from the function.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace NLP.TextClassification
{
    2 references
    public class BayesianClassifier: TextClassifier
    {
        //Dictionary<classLabel, Dictionary<word, count>>
        9 references
        public Dictionary<string, Dictionary<string, int>> classTokenCounts { get; private set; }

        //Dictionary<classLabel, count>
        8 references
        public Dictionary<string, int> classCounts { get; private set; }

        //HashSet<word>
        4 references
        public HashSet<string> vocabulary { get; private set; }
        4 references
        public int totalItems { get; private set; }

        //public override void Initialize(Vocabulary vocabulary)
        2 references
        public override void Initialize()
        {
            classTokenCounts = new Dictionary<string, Dictionary<string, int>>();
            classCounts = new Dictionary<string, int>();
            vocabulary = new HashSet<string>();
            totalItems = 0;
        }
```

In the figure above, BayesianClassifier inherits TextClassifier. A Dictionary<classLabel, Dictionary<word, count>> classTokenCounts is created to get the words belonging in a class and their counts. A Dictionary<classLabel, count> is created to get the number of words in a class. A HashSet<word> vocabulary is created to get the words more easily. totalItems is used to find the total number of items in the dataset.
In the Initialize() function, all the data structures above are initialised and totalItems is set to 0

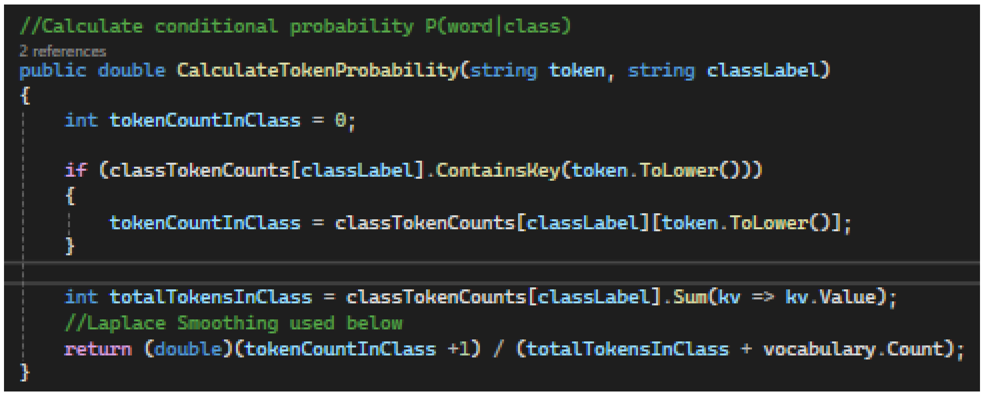


```
1 reference
public void Train(List<TextClassificationDataItem> dataSet)
{
    foreach (var item in dataSet)
    {
        //Count the class occurrences
        if (!classCounts.ContainsKey(item.ClassLabel.ToString()))
        {
            classCounts[item.ClassLabel.ToString()] = 0;
            classTokenCounts[item.ClassLabel.ToString()] = new Dictionary<string, int>();
        }
        classCounts[item.ClassLabel.ToString()]++;
        totalItems++;

        //Count the token occurrences for each class
        foreach (var token in item.TokenList)
        {
            string tokenSpelling = token.Spelling.ToLower();
            vocabulary.Add(tokenSpelling);
            if (!classTokenCounts[item.ClassLabel.ToString()].ContainsKey(tokenSpelling))
            {
                classTokenCounts[item.ClassLabel.ToString()][tokenSpelling] = 0;
            }
            classTokenCounts[item.ClassLabel.ToString()][tokenSpelling]++;
        }

    }
}
2 references
public override int Classify(List<Token> tokenList)
{
    double maxProbability = double.MinValue;
    string bestClass = null;

    foreach (var classLabel in classCounts.Keys)
    {
        double classProbability = CalculateClassProbability(classLabel);

        double logProbability = Math.Log(classProbability);
        foreach (var token in tokenList)
        {
            double tokenProbability = CalculateTokenProbability(token.Spelling, classLabel);
            logProbability += Math.Log(tokenProbability);
        }
        if (logProbability > maxProbability)
        {
            maxProbability = logProbability;
            bestClass = classLabel;
        }
    }
    return int.Parse(bestClass);
}

// Calculate prior probability P(class)
2 references
public double CalculateClassProbability(string classLabel)
{
    return (double)classCounts[classLabel] / totalItems;
}
```

In the figure above, the Train(List<TextClassificationDataItem> dataSet) function initializes the inner dictionary of classTokenCounts and sets the count in classCounts to 0 for that class, if the class is not in classCounts. The totalItems and the classCount is incremented.
Every token in TokenList is then added to the vocabulary. If the token is not found in classTokenCount, the count is set to 0. The count is then incremented.

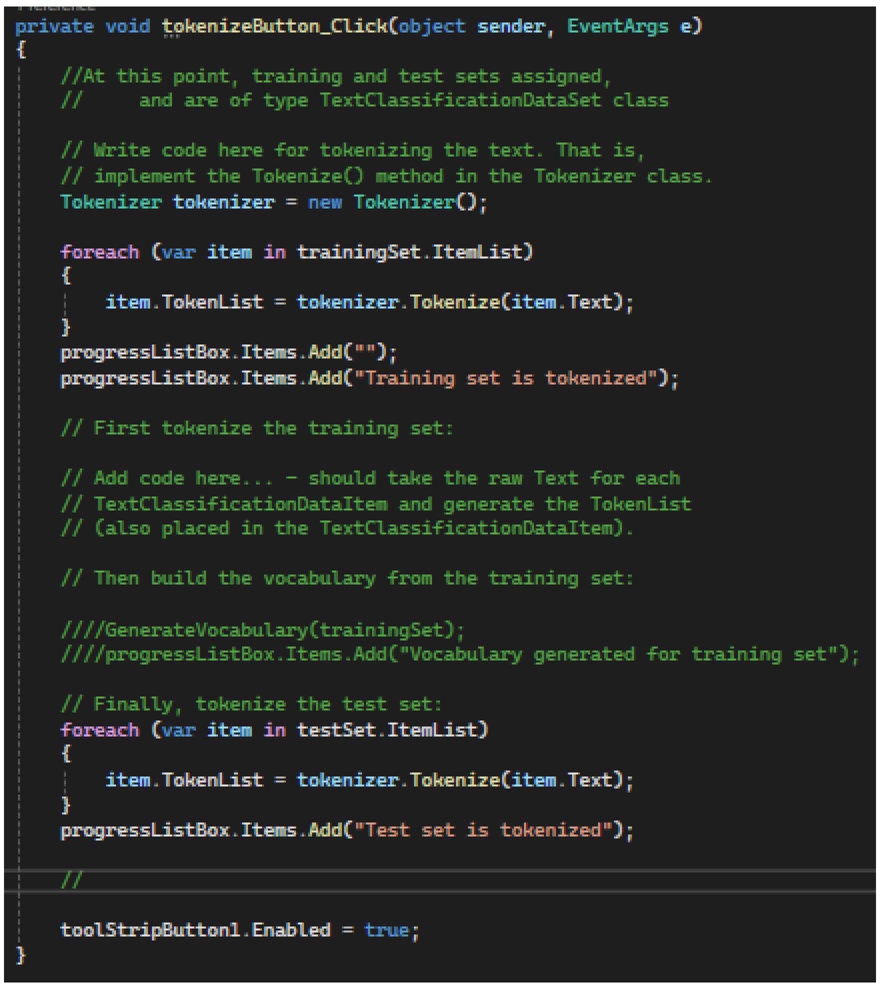
The int Classify(List<Token> tokenList) function determines the classProbabilities of each class and logs them to get larger values. The token probability is then determined for each token and also logged to get larger values. The two log values are summed and compared to the maxProbability. If the logProbability is larger than the maxProbability, then maxProbability is reassigned to logProbability and the bestClass is the classLabel with the new maxProbability. The bestClass is then returned by the function.

The double CalculateClassProbability(string classLabel) function returns the class probability by dividing the number of words in a class with the total number of items.



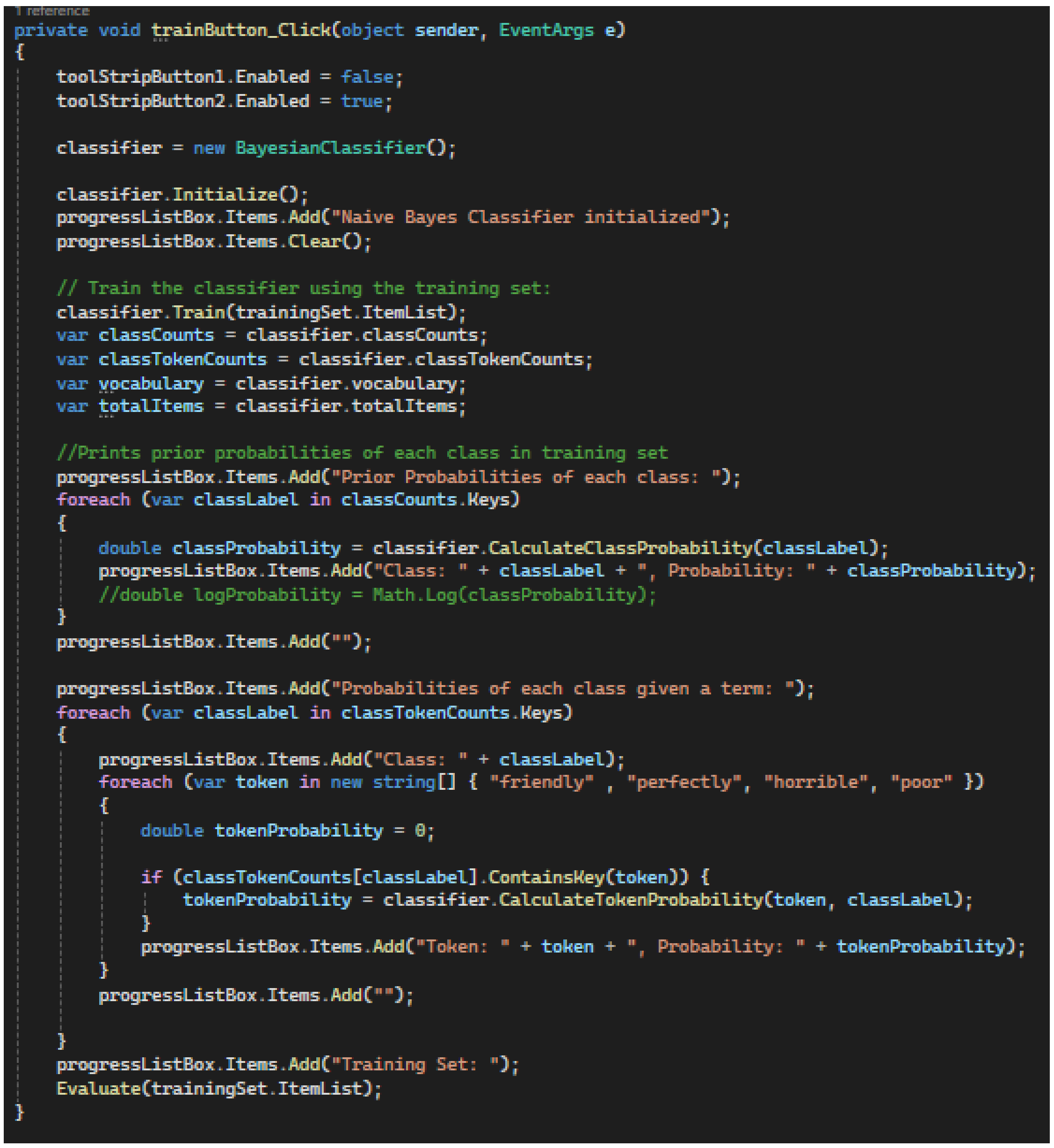
```
//Calculate conditional probability P(word|class)
2 references
public double CalculateTokenProbability(string token, string classLabel)
{
    int tokenCountInClass = 0;

    if (classTokenCounts[classLabel].ContainsKey(token.ToLower()))
    {
        tokenCountInClass = classTokenCounts[classLabel][token.ToLower()];
    }

    int totalTokensInClass = classTokenCounts[classLabel].Sum(kv => kv.Value);
    //Laplace Smoothing used below
    return (double)(tokenCountInClass +1) / (totalTokensInClass + vocabulary.Count);
}
```

In the figure above, the CalculateTokenProbability function returns a double and has (string token, string classLabel) as its parameters. Given the classLabel, the number of a certain token is determined. The total number of tokens in that class is then found. Using the Naive Bayes formula and Laplace smoothing, the token probability is calculated.

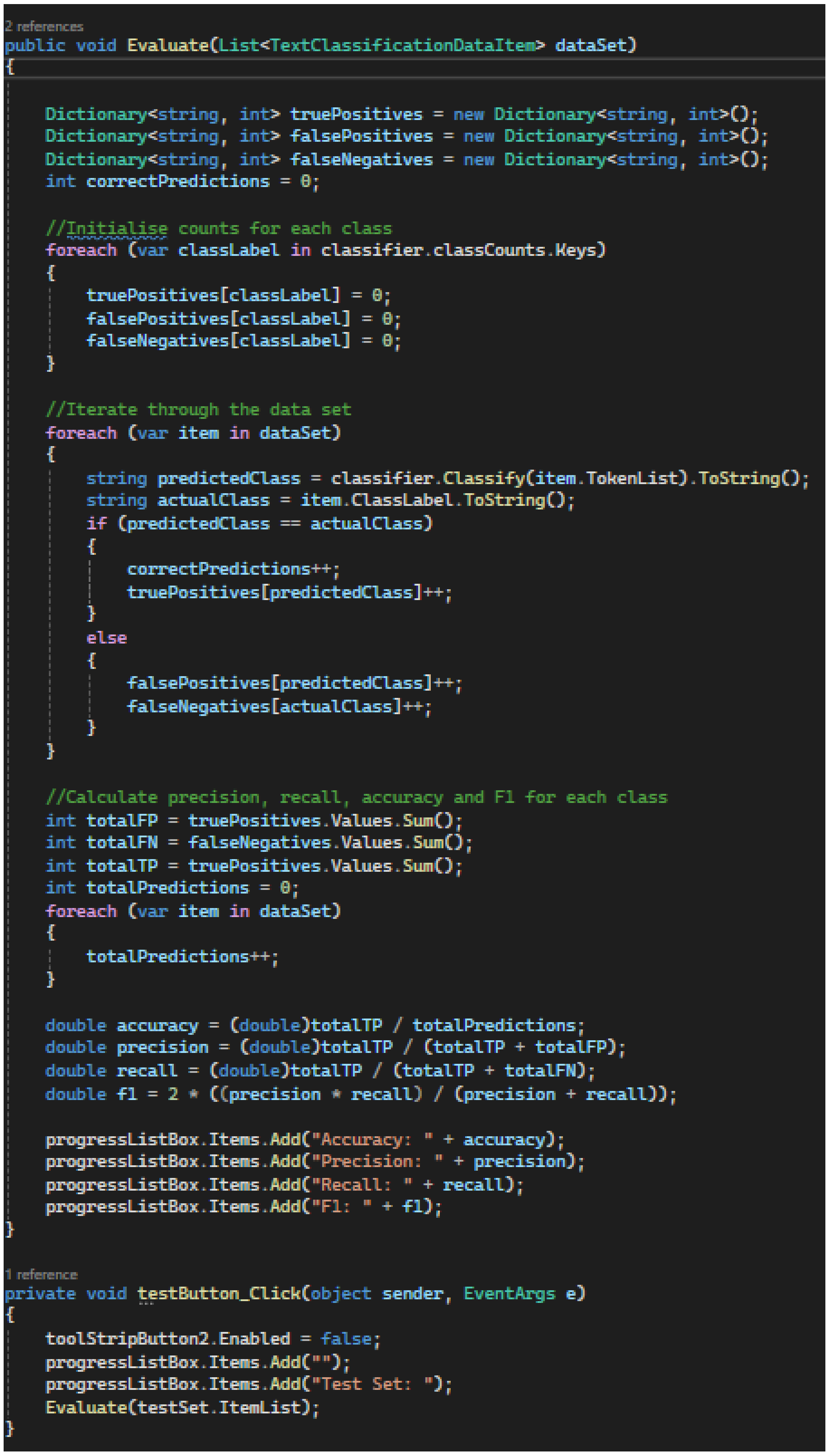


```
private void tokenizeButton_Click(object sender, EventArgs e)
{
    //At this point, training and test sets assigned,
    //     and are of type TextClassificationDataSet class

    // Write code here for tokenizing the text. That is,
    // implement the Tokenize() method in the Tokenizer class.
    Tokenizer tokenizer = new Tokenizer();

    foreach (var item in trainingSet.ItemList)
    {
        item.TokenList = tokenizer.Tokenize(item.Text);
    }
    progressListBox.Items.Add("");
    progressListBox.Items.Add("Training set is tokenized");

    // First tokenize the training set:

    // Add code here... - should take the raw Text for each
    // TextClassificationDataItem and generate the TokenList
    // (also placed in the TextClassificationDataItem).

    // Then build the vocabulary from the training set:

    ////GenerateVocabulary(trainingSet);
    ////progressListBox.Items.Add("Vocabulary generated for training set");

    // Finally, tokenize the test set:
    foreach (var item in testSet.ItemList)
    {
        item.TokenList = tokenizer.Tokenize(item.Text);
    }
    progressListBox.Items.Add("Test set is tokenized");

    //

    toolStripButton1.Enabled = true;
}
```

In the tokenizeButton_Click function, a Tokenizer is initialized when the “Tokenize” button is clicked. The training and test sets are then passed through the tokenizer


```
1 reference
private void trainButton_Click(object sender, EventArgs e)
{
    toolStripButton1.Enabled = false;
    toolStripButton2.Enabled = true;

    classifier = new BayesianClassifier();

    classifier.Initialize();
    progressListBox.Items.Add("Naive Bayes Classifier initialized");
    progressListBox.Items.Clear();

    // Train the classifier using the training set:
    classifier.Train(trainingSet.ItemList);
    var classCounts = classifier.classCounts;
    var classTokenCounts = classifier.classTokenCounts;
    var vocabulary = classifier.vocabulary;
    var totalItems = classifier.totalItems;

    //Prints prior probabilities of each class in training set
    progressListBox.Items.Add("Prior Probabilities of each class: ");
    foreach (var classLabel in classCounts.Keys)
    {
        double classProbability = classifier.CalculateClassProbability(classLabel);
        progressListBox.Items.Add("Class: " + classLabel + ", Probability: " + classProbability);
        //double logProbability = Math.Log(classProbability);
    }
    progressListBox.Items.Add("");

    progressListBox.Items.Add("Probabilities of each class given a term: ");
    foreach (var classLabel in classTokenCounts.Keys)
    {
        progressListBox.Items.Add("Class: " + classLabel);
        foreach (var token in new string[] { "friendly" , "perfectly", "horrible", "poor" })
        {
            double tokenProbability = 0;

            if (classTokenCounts[classLabel].ContainsKey(token)) {
                tokenProbability = classifier.CalculateTokenProbability(token, classLabel);
            }
            progressListBox.Items.Add("Token: " + token + ", Probability: " + tokenProbability);
        }
        progressListBox.Items.Add("");

    }
    progressListBox.Items.Add("Training Set: ");
    Evaluate(trainingSet.ItemList);
}
```

In the figure above, the trainButton_Click is triggered when the “Train” button is clicked. The BayesianClassifier classifier is initialized. The Initialize() function is called to set the values of the BayesianClassifier classifier. The Train function is called to train the classifier. The classCount, classTokenCounts, vocabulary and totalItems are obtained from the classifier. The probability of each class is then calculated using the CalculateClassProbability function. The token probability is then counted for the words “friendly”,”perfectly”,”poor” and “horrible” using the CalculateTokenProbability function. The performance of the training set is determined using the Evaluate function.

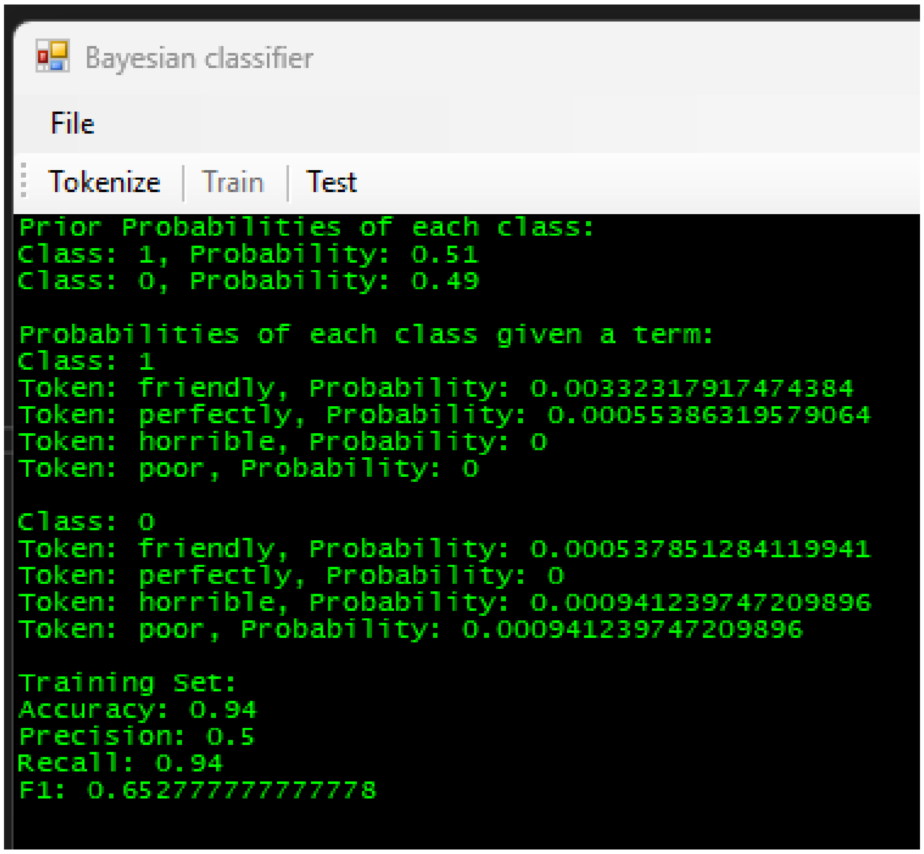


```
2 references
public void Evaluate(List<TextClassificationDataItem> dataSet)
{

    Dictionary<string, int> truePositives = new Dictionary<string, int>();
    Dictionary<string, int> falsePositives = new Dictionary<string, int>();
    Dictionary<string, int> falseNegatives = new Dictionary<string, int>();
    int correctPredictions = 0;

    //Initialise counts for each class
    foreach (var classLabel in classifier.classCounts.Keys)
    {
        truePositives[classLabel] = 0;
        falsePositives[classLabel] = 0;
        falseNegatives[classLabel] = 0;
    }

    //Iterate through the data set
    foreach (var item in dataSet)
    {
        string predictedClass = classifier.Classify(item.TokenList).ToString();
        string actualClass = item.ClassLabel.ToString();
        if (predictedClass == actualClass)
        {
            correctPredictions++;
            truePositives[predictedClass]++;
        }
        else
        {
            falsePositives[predictedClass]++;
            falseNegatives[actualClass]++;
        }
    }

    //Calculate precision, recall, accuracy and F1 for each class
    int totalFP = truePositives.Values.Sum();
    int totalFN = falseNegatives.Values.Sum();
    int totalTP = truePositives.Values.Sum();
    int totalPredictions = 0;
    foreach (var item in dataSet)
    {
        totalPredictions++;
    }

    double accuracy = (double)totalTP / totalPredictions;
    double precision = (double)totalTP / (totalTP + totalFP);
    double recall = (double)totalTP / (totalTP + totalFN);
    double f1 = 2 * ((precision * recall) / (precision + recall));

    progressListBox.Items.Add("Accuracy: " + accuracy);
    progressListBox.Items.Add("Precision: " + precision);
    progressListBox.Items.Add("Recall: " + recall);
    progressListBox.Items.Add("F1: " + f1);
}

1 reference
private void testButton_Click(object sender, EventArgs e)
{
    toolStripButton2.Enabled = false;
    progressListBox.Items.Add("");
    progressListBox.Items.Add("Test Set: ");
    Evaluate(testSet.ItemList);
}
```

In the figure above, the Evaluate(List<TextClassificationDataItem> dataSet) function creates three dictionary<classLabel, count> pairs, one each for truePositives, falsePositives and falseNegatives.
The correctPredictions is initialised to 0
The counts for each class for the three dictionaries are initialised to 0
For each item in the dataSet, a predictedClass is obtained using Classify(TokenList) method and the actualClass is obtained via the item's ClassLavel
If the predictedClass matches the actualClass, the correctPredictions is incremented alongside truePositives. If not, the falsePositives and the falseNegatives are incremented
The accuracy, recall, precision and f1 are calculated using truePositives, falsePositives and falseNegatives

testButton_Click executes Evaluate function on the test set

Results of Restaurant Reviews

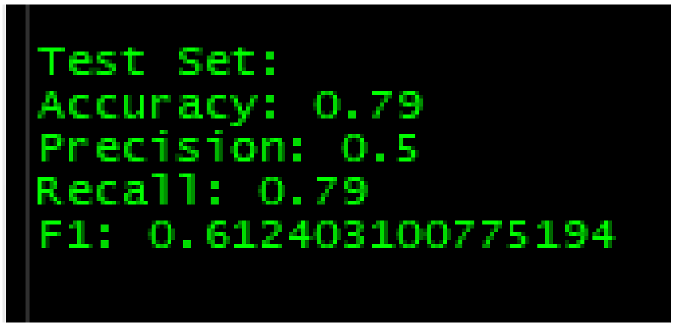


```
Prior Probabilities of each class:
Class: 1, Probability: 0.51
Class: 0, Probability: 0.49

Probabilities of each class given a term:
Class: 1
Token: friendly, Probability: 0.00332317917474384
Token: perfectly, Probability: 0.00055386319579064
Token: horrible, Probability: 0
Token: poor, Probability: 0

Class: 0
Token: friendly, Probability: 0.000537851284119941
Token: perfectly, Probability: 0
Token: horrible, Probability: 0.000941239747209896
Token: poor, Probability: 0.000941239747209896

Training Set:
Accuracy: 0.94
Precision: 0.5
Recall: 0.94
F1: 0.652777777777778
```

The prior probability for the training set for class 1 and 0 are 0.51 and 0.49 respectively.

In 5 decimal places:
Given the term friendly, it has a probability of 0.00332 and 0.00054 for class 1 and 0 respectively.
Given the term perfectly, it has a probability of 0.00055 and 0 for class 1 and 0 respectively.
Given the term horrible, it has a probability of 0 and 0.00094 for class 1 and 0 respectively.
Given the term poor, it has a probability of 0 and 0.00094 for class 1 and 0 respectively.
It seems that words with negative connotations like horrible and poor are not associated with a positive class since their probability in class 1 is 0.
Also, perfectly, a word with positive connotations is not associated with a negative class since their probability in class 0 is 0
As for friendly, the dataset may not be large enough to properly classify the word as positive.

For the training set, it has an accuracy of 0.94, precision of 0.5, recall of 0.94 and F1 measure of 0.65(rounded of to 2 decimal places)

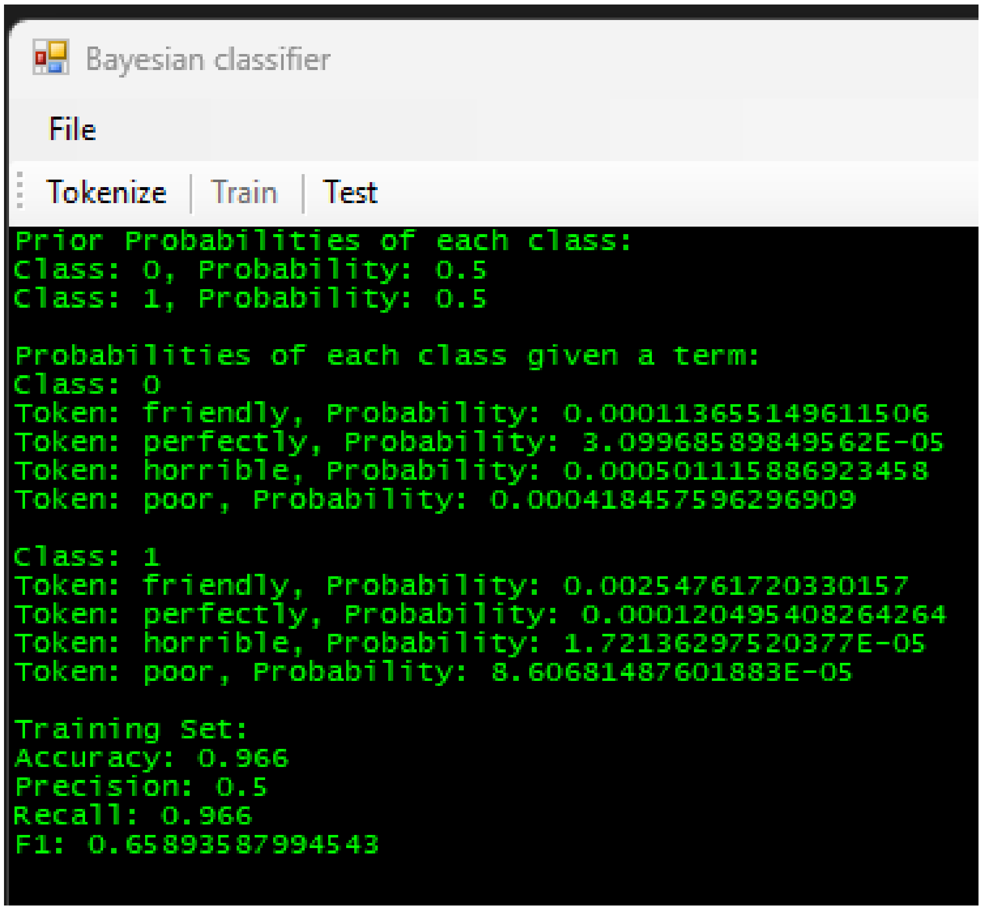


```
Test Set:
Accuracy: 0.79
Precision: 0.5
Recall: 0.79
F1: 0.612403100775194
```

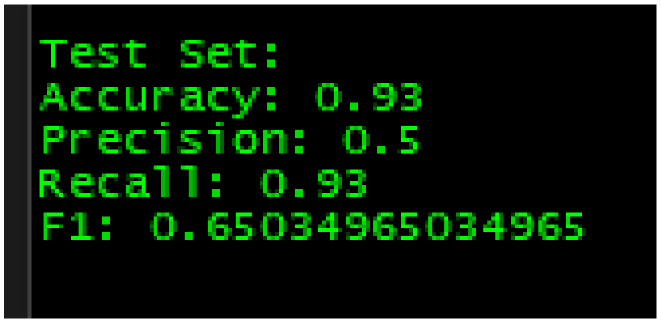
For the test set, it has an accuracy of 0.79, precision of 0.5, recall of 0.79 and F1 measure of 0.61(rounded of to 2 decimal places)

The performance measures are poor for the test set as there are unseen words in the test set. These words that are not seen in training have a probability of 0, making the whole class probability 0.

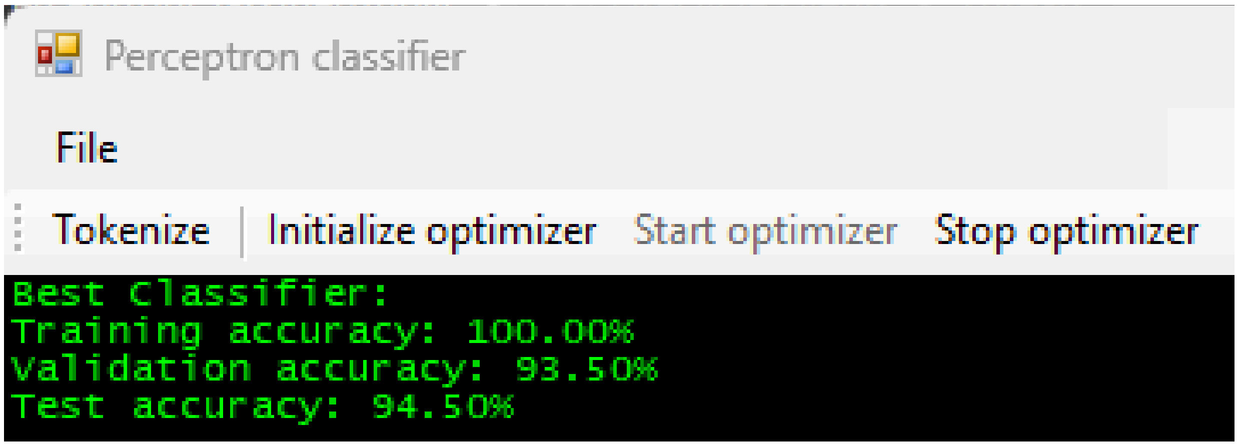
<u>Results of Airline Reviews</u>



```
Prior Probabilities of each class:
Class: 0, Probability: 0.5
Class: 1, Probability: 0.5

Probabilities of each class given a term:
Class: 0
Token: friendly, Probability: 0.000113655149611506
Token: perfectly, Probability: 3.09968589849562E-05
Token: horrible, Probability: 0.000501115886923458
Token: poor, Probability: 0.000418457596296909

Class: 1
Token: friendly, Probability: 0.00254761720330157
Token: perfectly, Probability: 0.000120495408264264
Token: horrible, Probability: 1.72136297520377E-05
Token: poor, Probability: 8.60681487601883E-05

Training Set:
Accuracy: 0.966
Precision: 0.5
Recall: 0.966
F1: 0.658935879945 43
```



```
Test Set:
Accuracy: 0.93
Precision: 0.5
Recall: 0.93
F1: 0.65034965034965
```



```
Best Classifier:
Training accuracy: 100.00%
Validation accuracy: 93.50%
Test accuracy: 94.50%
```

From the results above, we can see that the Perceptron classifier has a greater test accuracy of 94.5% than that of the Naive Bayes classifier of 93%.

Feature Independence is assumed in Naive Bayes. This assumption is often violated, resulting in suboptimal decision boundaries. Perceptrons do not have this assumption

A perceptron learns weights through iterative updates and adjusts to the training data. On the other hand, Naive Bayes only computes probabilities from frequency counts and does not generalise well if the dataset has complex relationships.