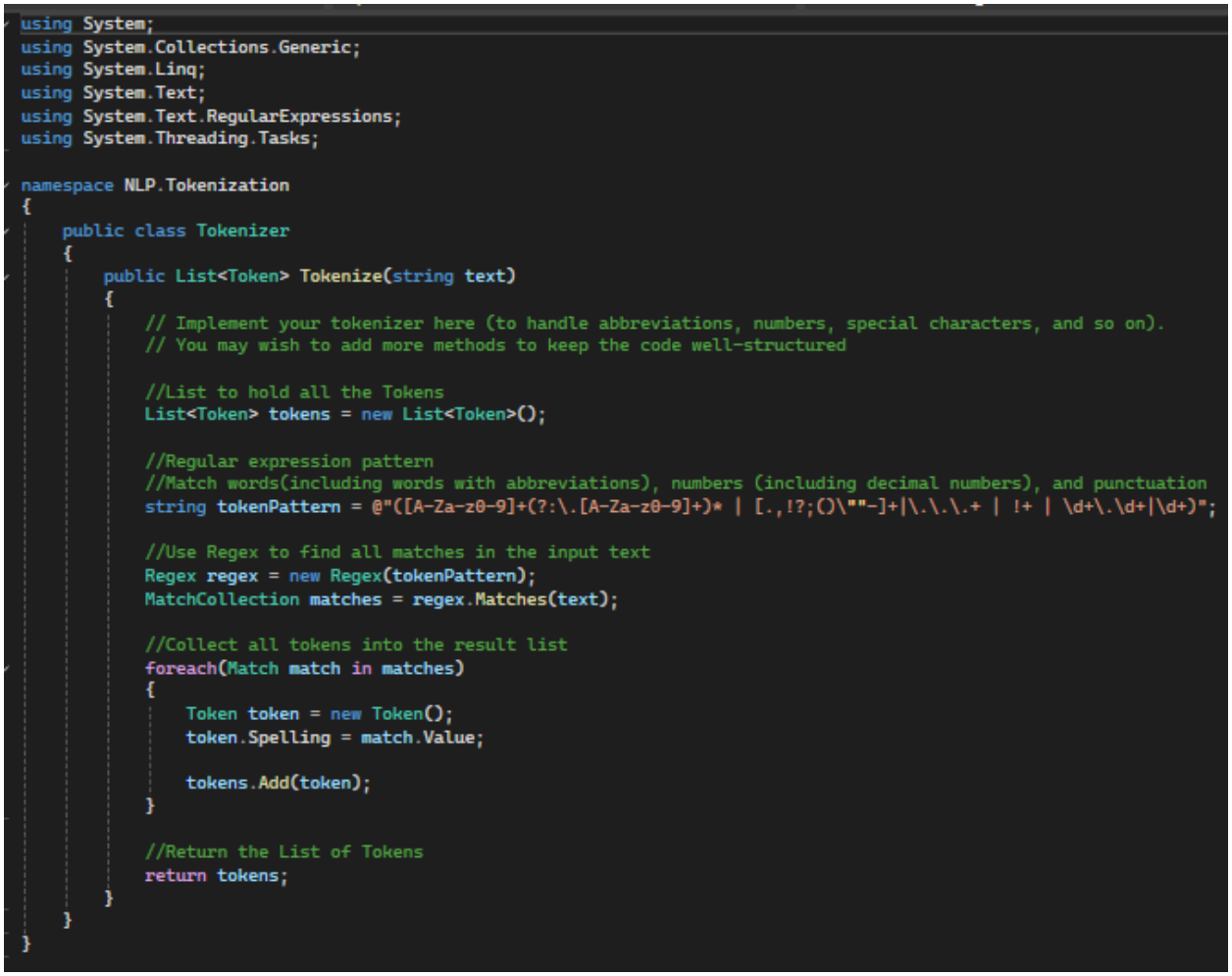
Explanation of C# implementation



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

namespace NLP.Tokenization
{
    public class Tokenizer
    {
        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 = @"([A-Za-z0-9]+(?:\.[A-Za-z0-9]+)* | [.,!?;()\""-]+|\.\.\.+ | !+ | \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;

                tokens.Add(token);
            }

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

The figure above is the Tokenizer class. It has a function Tokenize that has parameters string Text and return value of List<Token>. Using Regex, words with abbreviations, numbers and punctuations are found. They are then added to a List of Tokens as a Token.

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

namespace NLP.TextClassification
{
    public class PerceptronClassifier: TextClassifier
    {
        private double bias;
        // The Key (string) should identify (the spelling of) the token
        private Dictionary<string, double> weightDictionary;

        public override void Initialize(Vocabulary vocabulary)
        {
            // Write this method, setting up the weight dictionary
            // (one key-value pair (string-double) for each token in
            // the vocabulary, where the value represents the weight of
            // the corresponding token in the perceptron classifier.

            // Initially, assign random weights in the range [-1,1].
            // To obtain random numbers (in [0,1[) use the Random class,
            // with a suitable (integer) random number seed.

            Random random = new Random(42);     //Random Number Seed = 42

            weightDictionary = new Dictionary<string, double>();
            bias = 0;

            foreach (string tokenSpelling in vocabulary.getVocabulary().Keys)
            {
                double randomWeight = random.NextDouble() * 2 -1;

                weightDictionary[tokenSpelling] = randomWeight;
            }
        }

        public override int Classify(List<Token> tokenList)
        {
            // ToDo: Write this method

            // The input should be the tokens of
            // the text (i.e., a TextClassificationDataItem) that is being classified.


            // Remove the line below - needed for compilation of this skeleton code,
            // since the method must return an integer.
            // The returned integer should be the class ID (in this case, either 0 or 1).

            double sum = bias;

            foreach (Token token in tokenList)
            {
                if (weightDictionary.ContainsKey(token.Spelling))
                {
                    sum += weightDictionary[token.Spelling];
                }
            }

            return sum >= 0 ? 1 :0;

        }
```

The figure above shows the Perceptron class. A bias value and a weightDictionary<word, weight> is created. In Initialize(Vocabulary vocabulary), random number seed 42 is used to assign random weight of range [-1,1] to each word of the weightDictionary

In int Classify(List<Token> tokenList), a class ID is assigned to each token. If the sum, which is the summation of itself and the weight of a token, is more than 0, it is assigned 1, or else it is assigned 0.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Text;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;
using NLP;
using NLP.TextClassification;
using NLP.Tokenization;

namespace PerceptronClassifierApplication
{
    public partial class MainForm : Form
    {
        private const string TEXT_FILE_FILTER = "Text files (*.txt)|*.txt";

        private PerceptronClassifier classifier = null;
        private Vocabulary vocabulary = null;
        private TextClassificationDataSet trainingSet = null;
        private TextClassificationDataSet validationSet = null;
        private TextClassificationDataSet testSet = null;

        private Task optimizerTask; //Task for running optimizer
        private CancellationTokenSource cancellationTokenSource;
        private PerceptronClassifier bestClassifier;

        private int epochs = 200;
        private double learningRate = 0.1;
        private double bestvalidationAccuracy = 0.0;

        private StreamWriter writer;

        TextClassificationDataItem correctExample;
        TextClassificationDataItem incorrectExample;
```

A Task and CancellationTokenSource object are created to handle the task of running the optimiser and control when it is stopped. A PerceptronClassifier object is created to keep information of the best classifier
The epochs is set to 200 such that results can be displayed on the graph even though it is capable to run continuously if the epoch is increased to a significantly large number
The learningRate of the optimiser is set as 0.1 and the best validation accuracy is initialised as 0.0
A StreamWriter object is created to write the training and validation accuracies of every epoch to a txt file

Two TextClassificationDataItem objects are created for a correctly classified sentence and an incorrectly classified sentence.

```
private void GenerateVocabulary(TextClassificationDataSet dataSet)
{
    // Write a method that generates the vocabulary. Note that this
    // should ONLY be done for the training set!

    //Pass in TextClassificationnDataSet as parameter
    //Return Dictionary<string, Token>

    vocabulary = new Vocabulary();
    vocabulary.vocabularize(dataSet);

    // You must generate an instance of the Vocabulary class,
    // which you must also implement (a skeleton is available
    // in the NLP library)
}
```

In the figure above, the GenerateVocabulary(TextClassificationDataSet dataSet) function initialised the vocabulary object of the Vocabulary class. The vocabularize function is then called, passing dataSet as the parameter.

```
using System;
using System.Collections.Generic;
using System.Diagnostics.Tracing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using NLP.TextClassification;

namespace NLP
{
    public class Vocabulary
    {
        //Store the vocabulary dictionary
        private Dictionary<string, Token> vocab;

        //Constructor to initialize the dictionary
        public Vocabulary()
        {
            vocab = new Dictionary<string, Token>();
        }

        //Dictionary of <sentence, word>
        public void vocabularize(TextClassificationDataSet txds)
        {   //Pass in TextClassificationnDataSet as parameter
            //Return Dictionary<string, Token>

            foreach (TextClassificationDataItem txdi in txds.ItemList)
            {
                foreach (Token tk in txdi.TokenList)
                {
                    //Get the word/spelling
                    string word = tk.Spelling;

                    //Assign word as key and Token as value
                    if (!vocab.ContainsKey(word))
                    {
                        vocab[word] = tk;
                    }
                }
            }
        }

        //Method to get the vocabulary dictionary
        public Dictionary<string, Token> getVocabulary()
        {
            return vocab;
        }

        // Write this class - it should contain a data structure
        // that holds all the words in the vocabulary
        // Use a Dictionary<string, Token> for this purpose.

        // NOTE: In Problem1.1, the Vocabulary class is not used,
        // instead (only for that problem) the vocabulary is represented
        // as a simple List<Token>.
    }
}
```

In the figure above, a dictionary of <word, Token> is created. It is then assigned respective values using vocabularize(TextClassificationDataSet txds). A getVocabulary() function is also implemented for the vocabulary to be retrieved for future uses.

```
private void tokenizeButton_Click(object sender, EventArgs e)
{//At this point, training, test and validation 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");

    // Next, tokenize the validation set:
    foreach (var item in validationSet.ItemList)
    {
        item.TokenList = tokenizer.Tokenize(item.Text);
    }
    progressListBox.Items.Add("Validation set is tokenized");

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

    //
    initializeOptimizerButton.Enabled = true;
}
```

In the figure above, when the ‘Tokenize’ button is clicked, this function is triggered. A Tokenizer object is created. The training, validation and test sets are then tokenized using it. The vocabulary of the training set is also generated.

```
private void startOptimizerButton_Click(object sender, EventArgs e)
{
    startOptimizerButton.Enabled = false;
    stopOptimizerButton.Enabled = true;
    cancellationTokenSource = new CancellationTokenSource();
    CancellationToken canceltoken = cancellationTokenSource.Token;

    // Start the optimizer here.

    // For every epoch, the optimizer should (after the epoch has been completed)
    // trigger an event that prints the current accuracy (over the training set
    // and the validation set) of the perceptron classifier (in a thread-safe
    // manner, and with proper (clear) formatting). Do *not* involve
    // the test set here.

    //Specify file path to save the validation and training accuracy of every epoch
    string filePath = "C:\\Users\\liauz\\OneDrive\\Desktop\\Intelligent Agents\\Assignment 1\\Training and Validation Accuracy for Graph.txt";

    writer = new StreamWriter(filePath, false); //false overwrites the file if it already exists
    writer.WriteLine("Epoch\tTraining Accuracy\tValidation Accuracy");

    optimizerTask = Task.Run(() =>
    {
        if (!progressListBox.IsDisposed)
        {
            progressListBox.Invoke((MethodInvoker)(() => progressListBox.Items.Add("")));
        }
        //Iterate through each epoch
        for (int epoch = 0; epoch < epochs; epoch++)
        {

            //Iterate through each item in the training set
            foreach (var item in trainingSet.ItemList)
            {
                //Classify the item
                int predicted = classifier.Classify(item.TokenList);
                int actual = item.ClassLabel;
                //If the classification is incorrect
                if (predicted != actual)
                {
                    double error = actual - predicted;
                    //Update the weights
                    foreach (Token token in item.TokenList)
                    {
                        if (classifier.WeightDictionary.ContainsKey(token.Spelling))
                        {
                            classifier.WeightDictionary[token.Spelling] += learningRate * error;
                        }

                    }

                    //Update bias
                    classifier.Bias += learningRate * error;
                }
            }

            //Log accuracy in a thread safe way
            double trainingAccuracy = Evaluate(trainingSet.ItemList, classifier);
            double validationAccuracy = Evaluate(validationSet.ItemList, classifier);

            writer.WriteLine((epoch+1) + "\t" + trainingAccuracy + "\t" + validationAccuracy);
```

```
            // Check if the validation accuracy is the best so far
            if (validationAccuracy > bestvalidationAccuracy)
            {
                bestvalidationAccuracy = validationAccuracy;
                // Save the classifier to a file

                bestClassifier = (PerceptronClassifier)classifier.Copy();
            }

            this.Invoke((MethodInvoker)(() =>
            {
                progressListBox.Items.Add("Epoch: " + (epoch + 1) + " Training accuracy: " + trainingAccuracy.ToString("P") + " Validation accuracy: " + validationAccuracy.ToString("P"));

            }));

            if (canceltoken.IsCancellationRequested)
            {
                break;
            }
        }
        //Flush and Close the writer
        writer.Flush();
        writer.Close();
    }, canceltoken);
}
```

In the figure above, this function is triggered when the “Start Optimiser” button is pressed. A CancellationTokenSource object is initialised and a cancelToken is obtained. A filepath is

specified for saving the training and validation accuracies of every epoch. A StreamWriter object is also initialised to write to filepath and overwrite it if it already exists. The Task object optimizerTask then runs the rest over most of the remaining parts of the function. The predicted class is obtained using the Classify function. It is then compared to the actual class. If they do not match, the weight of the word is updated using the learning rate and the error, that is found from the difference of the actual and predicted values. The bias is also updated with the learning rate and the error

The training and the validation accuracy of the current epoch is obtained using Evaluate(), explained below. A header is written to the file. The validationAccuracy is checked if it is greater than the bestValidationAccuracy and updates the bestValidationAccuracy if it is. bestClassifier is then assigned to the current classifier if the validationAccuracy is greater than the bestValidationAccuracy.

To print the results in a thread safe way, MethodInvoker and the Invoke function are used.

The function then checks if cancellation has been requested and exits the function if it has.

The StreamWriter object is then flushed and closed, saving the file containing the training and validation accuracies of every epoch

```
private async void stopOptimizerButton_Click(object sender, EventArgs e)
{
    stopOptimizerButton.Enabled = false;

    // Stop the optimizer here.
    if (cancellationTokenSource != null)
    {
        cancellationTokenSource.Cancel();
    }

    await Task.Delay(1000); //Wait for a second to ensure the optimizer has stopped

    //Evaluate the best classifier using the test set
    if (bestClassifier != null)
    {
        double bestClassifierTestAccuracy = EvaluateBest(testSet.ItemList, bestClassifier);
        double bestClassifierValidationAccuracy = Evaluate(validationSet.ItemList, bestClassifier);
        double bestClassifierTrainingAccuracy = Evaluate(trainingSet.ItemList, bestClassifier);

        ///Display the result in a thread safe way
        if (!progressListBox.IsDisposed)
        {
            progressListBox.Invoke((MethodInvoker)(() =>
            {
                progressListBox.Items.Clear();
                progressListBox.Items.Add("Best Classifier:");
                progressListBox.Items.Add("Training accuracy: " + bestClassifierTrainingAccuracy.ToString("P"));
                progressListBox.Items.Add("Validation accuracy: " + bestClassifierValidationAccuracy.ToString("P"));
                progressListBox.Items.Add("Test accuracy: " + bestClassifierTestAccuracy.ToString("P"));
                progressListBox.Items.Add("");


                if (correctExample != null && incorrectExample != null)
                {
                    progressListBox.Items.Add("Correctly Classified Example:");
                    progressListBox.Items.Add($"Sentence:");
                    PrintSentenceWithLineBreaks(string.Join(" ", correctExample.TokenList.Select(t => t.Spelling)));
                    progressListBox.Items.Add($"Actual Class: {correctExample.ClassLabel}, Predicted: {bestClassifier.Classify(correctExample.TokenList)}");
                    progressListBox.Items.Add("");

                    progressListBox.Items.Add("Incorrectly Classified Example:");
                    progressListBox.Items.Add($"Sentence:");
                    PrintSentenceWithLineBreaks(string.Join(" ", incorrectExample.TokenList.Select(t => t.Spelling)));
                    progressListBox.Items.Add($"Actual Class: {incorrectExample.ClassLabel}, Predicted: {bestClassifier.Classify(incorrectExample.TokenList)}");

                }
                PrintTopBottomWords();
            }));
        }
    }
}
```

In the figure above, the function is triggered when the “Stop Optimizer” button is clicked. The cancellationTokenSource calls the Cancel function to stop the process of the running optimizer. A delay is created to allow for the optimizer to finish execution of its current iteration. With the bestClassifier previously assigned, the training, validation and test accuracies are determined

using the Evaluate and EvaluateBest functions, explained below. The results are then displayed in a thread safe way using MethodInvoker and Invoker functions. PrintSentenceWithLineBreaks, explained below too, formats the printing of the correctly and incorrectly classified sentences.

```
private void PrintSentenceWithLineBreaks(string sentence)
{
    //Split the sentence into individual words
    string[] words = System.Text.RegularExpressions.Regex.Split(sentence, @"[\s\t]+");

    //Create a new StringBuilder
    StringBuilder sb = new StringBuilder();

    //Iterate through words and print in groups of 15
    for (int i = 0; i < words.Length; i++)
    {
        sb.Append(words[i] + " ");
        if ((i % 15 == 0 && i!=0)|| i==words.Length - 1)
        {
            progressListBox.Items.Add(sb.ToString());
            //Reset the StringBuilder for the next group of words
            sb.Clear();
        }
    }


    // For simplicity (even though one may perhaps resume the optimizer), at this
    // point, evaluate the best classifier (= best validation performance) over
    // the *test* set, and print the accuracy to the screen (in a thread-safe
    // manner, and with proper (clear) formatting).

    stopOptimizerButton.Enabled = true; // A bit ugly, should wait for the
    // optimizer to actually stop, but that's OK, it will stop quickly.
}
```

In the figure above, PrintSentenceWithLineBreaks(string sentence) splits the text based on space and tabs. When printing the sentence onto the application, it goes to a new line for every 15 words. This is only used of formatting the display of the sentence on the application

```
private double EvaluateBest(List<TextClassificationDataItem> items, PerceptronClassifier model)
{
    if (items == null || model == null || items.Count == 0)
    {
        correctExample = null;
        incorrectExample = null;
        return 0.0;
    }

    int correct = 0;
    correctExample = null;
    incorrectExample = null;


    foreach (var item in items)
    {
        int predicted = classifier.Classify(item.TokenList);
        int actual = item.ClassLabel;
        if (predicted == actual)
        {
            correct++;
            if (correctExample == null)
            {
                correctExample = item;
            }
        }
        else
        {
            if (incorrectExample == null)
            {
                incorrectExample = item;
            }
        }
    }
    return (double)correct / items.Count;
}

private double Evaluate(List<TextClassificationDataItem> items, PerceptronClassifier model)
{
    if (items == null || model == null || items.Count == 0)
    {
        return 0.0;
    }

    int correct = 0;

    foreach (var item in items)
    {
        int predicted = classifier.Classify(item.TokenList);
        int actual = item.ClassLabel;
        if (predicted == actual)
        {
            correct++;
        }
    }
    return (double)correct / items.Count;
}
```

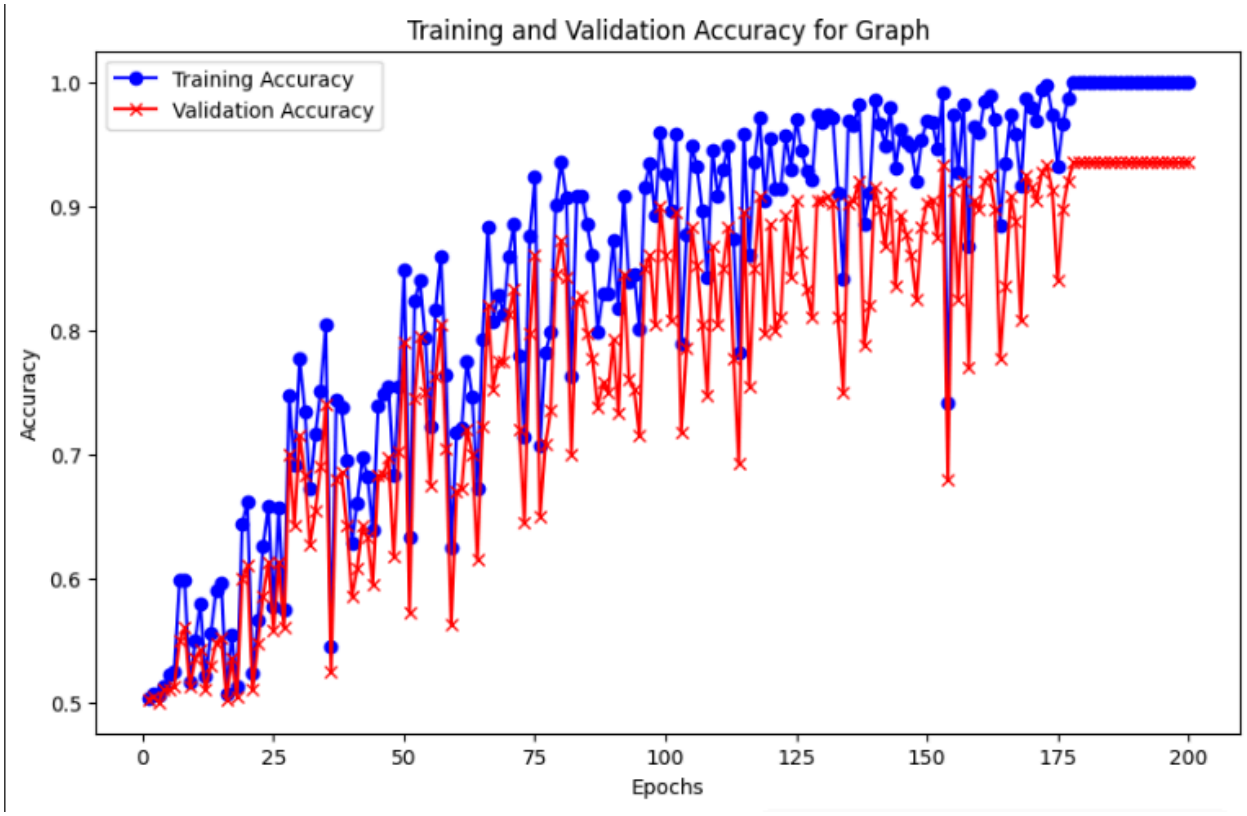
In the figure above, Evaluate(List<TextClassificationDataItem> item, PerceptronClassifier model) is used to obtain the accuracy of the model on classifying items. If there are no items or the model is non existent, then the accuracy is 0. Else, a predicted value is obtained using the Classify(TokenList) function. The predicted value is compared to the actual value to get the number of correctly classified items. A fraction is then obtained to get the accuracy.

EvaluateBest(List<TextClassificationDataItem> item, PerceptronClassifier model) is similar to Evaluate() but it also gets an incorrectly and a correctly classified sentence to assign to the global variables incorrectExample and correctExamples.

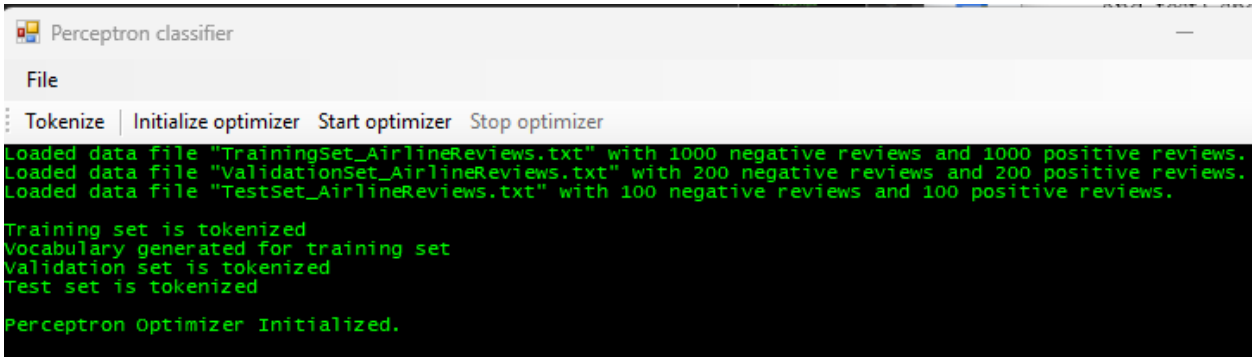
```
private void PrintTopBottomWords()
{
    if (bestClassifier == null || bestClassifier.WeightDictionary.Count == 0 || bestClassifier.WeightDictionary == null)
    {
        return;
    }

    //Sort the dictionary by weight value, descending order
    var sortedWeights = bestClassifier.WeightDictionary.OrderByDescending(x => x.Value).ToList();

    // Top 10 positive words
    var topPositiveWords = sortedWeights.Take(10).ToList();

    // Top 10 negative words
    var topNegativeWords = sortedWeights.Reverse<KeyValuePair<string,double>>().Take(10).ToList();

    // Display results
    progressListBox.Items.Add("");
    progressListBox.Items.Add("Top 10 Positive words:");
    foreach (var word in topPositiveWords)
    {
        progressListBox.Items.Add(word.Key + " : " + word.Value);
    }

    progressListBox.Items.Add("");
    progressListBox.Items.Add("Top 10 Negative words:");
    foreach (var word in topNegativeWords)
    {
        progressListBox.Items.Add(word.Key + " : " + word.Value);
    }
```

In the figure above, PrintTopBottomWords() checks if the bestClassifier exists and if there is a weightDictionary in it with values inside. The weightDictionary is then sorted based on descending weights. The top 10 most positive weighted words are then retrieved and placed in a list. The weightDictionary is then reversed and the top 10 most negative weighted words are obtained. They are then printed onto the application

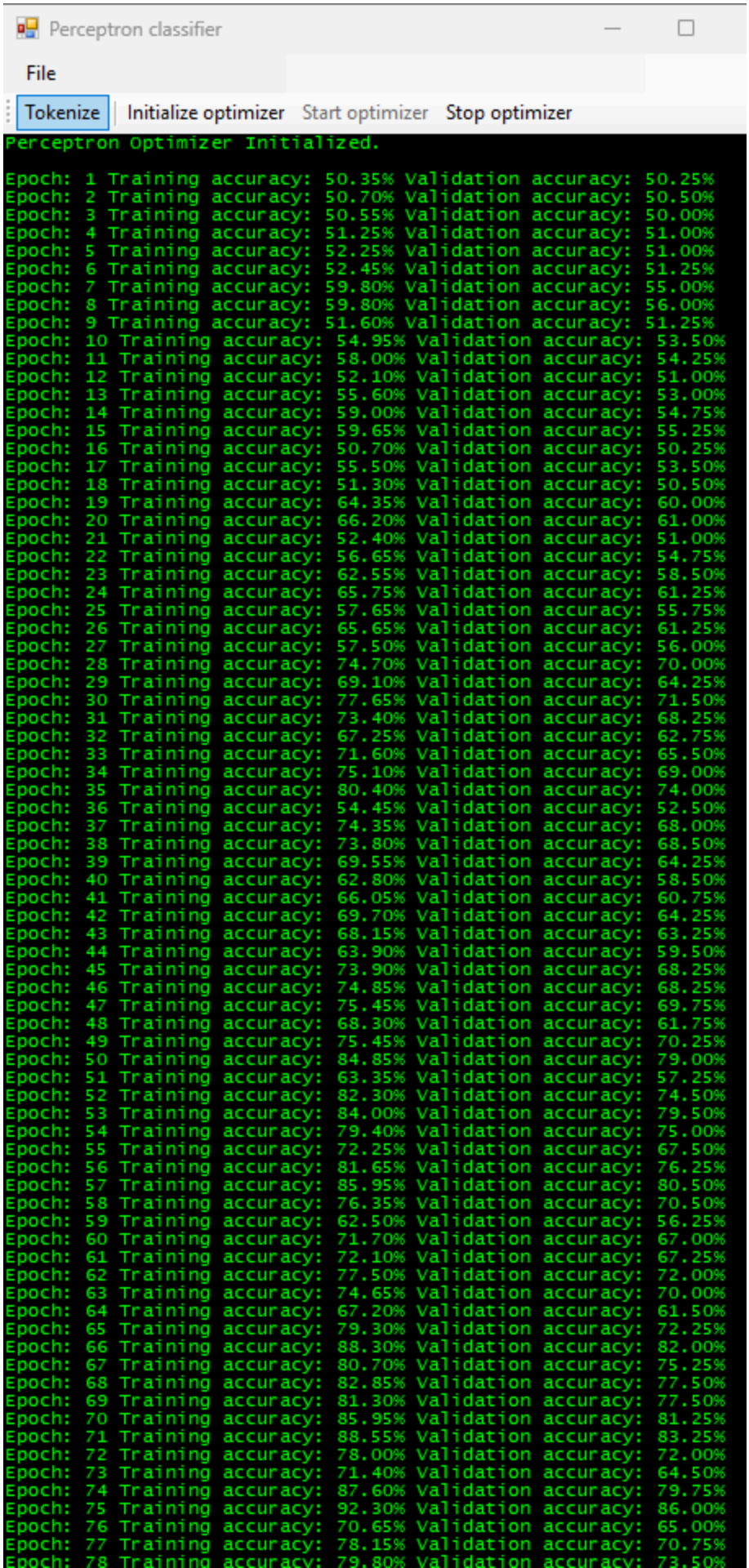
Results obtained and the Application Interface

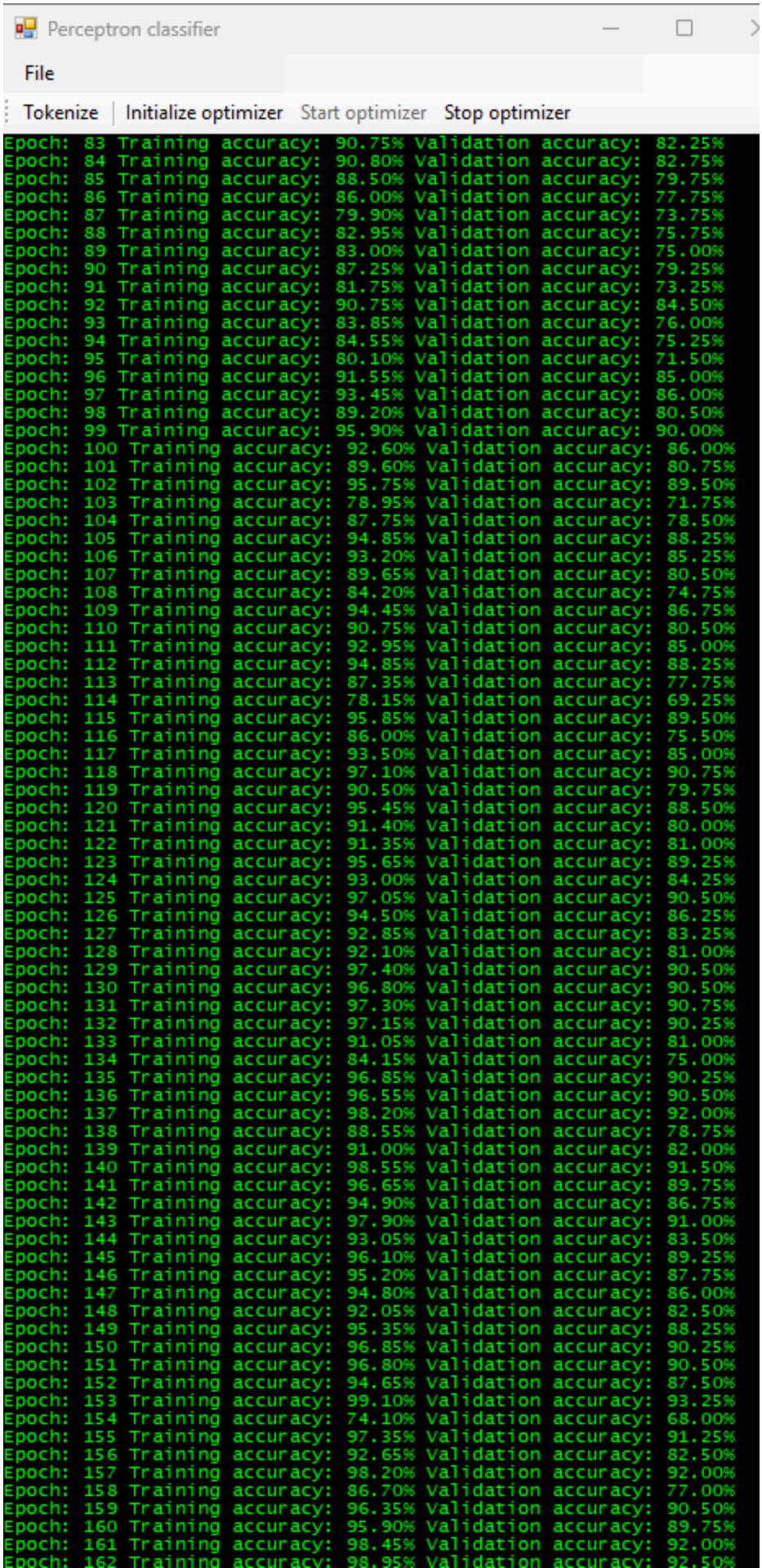


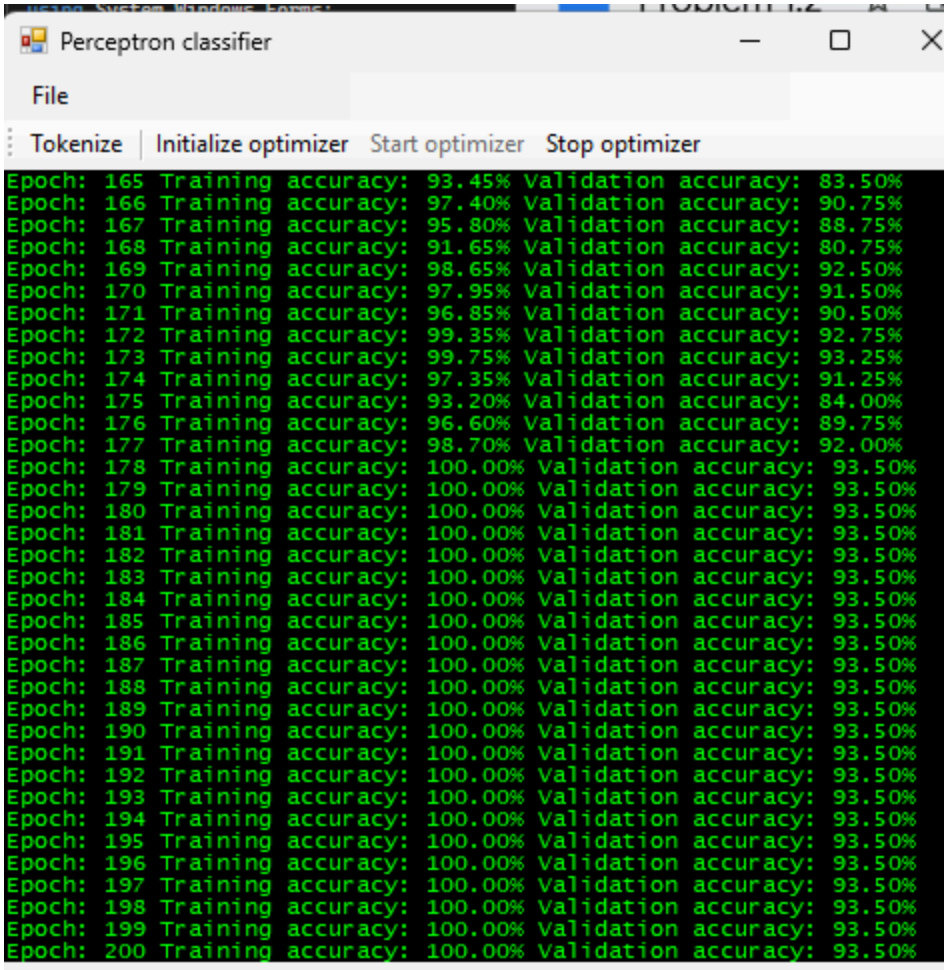
From the Figure above, we can see that as the number of epochs increases, the training and the validation accuracies both increase until they plateau at fixed accuracies. Also, we can see that the validation accuracy is always less than the training accuracy as the validation accuracy is based on the training done on the training set.



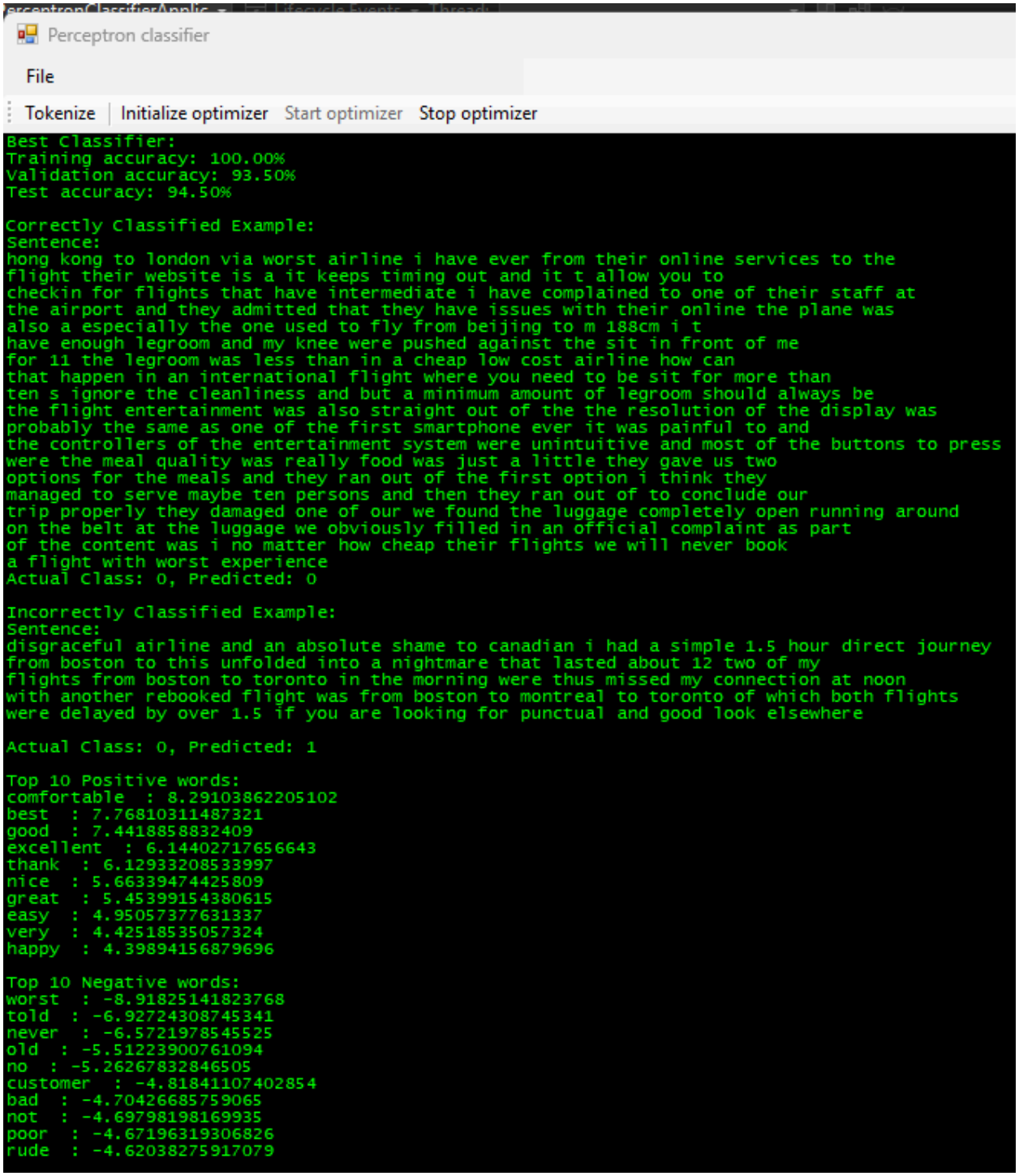
In the Figure above, the training, validation and the test sets are loaded into the application using their respective buttons found in ‘File’. Then the Training set is tokenized with a vocabulary generated for it using the ‘Tokenize’ button. After which, the validation and the test sets are tokenized too.Then the Perceptron Optimizer is Initialised using the ‘Initialise optimiser’ button


Perceptron classifier
File
Tokenize
Initialize optimizer
Start optimizer
Stop optimizer
Perceptron Optimizer Initialized.
Epoch: 1 Training accuracy: 50.35% Validation accuracy: 50.25%
Epoch: 2 Training accuracy: 50.70% Validation accuracy: 50.50%
Epoch: 3 Training accuracy: 50.55% Validation accuracy: 50.00%
Epoch: 4 Training accuracy: 51.25% Validation accuracy: 51.00%
Epoch: 5 Training accuracy: 52.25% Validation accuracy: 51.00%
Epoch: 6 Training accuracy: 52.45% Validation accuracy: 51.25%
Epoch: 7 Training accuracy: 59.80% Validation accuracy: 55.00%
Epoch: 8 Training accuracy: 59.80% Validation accuracy: 56.00%
Epoch: 9 Training accuracy: 51.60% Validation accuracy: 51.25%
Epoch: 10 Training accuracy: 54.95% Validation accuracy: 53.50%
Epoch: 11 Training accuracy: 58.00% Validation accuracy: 54.25%
Epoch: 12 Training accuracy: 52.10% Validation accuracy: 51.00%
Epoch: 13 Training accuracy: 55.60% Validation accuracy: 53.00%
Epoch: 14 Training accuracy: 59.00% Validation accuracy: 54.75%
Epoch: 15 Training accuracy: 59.65% Validation accuracy: 55.25%
Epoch: 16 Training accuracy: 50.70% Validation accuracy: 50.25%
Epoch: 17 Training accuracy: 55.50% Validation accuracy: 53.50%
Epoch: 18 Training accuracy: 51.30% Validation accuracy: 50.50%
Epoch: 19 Training accuracy: 64.35% Validation accuracy: 60.00%
Epoch: 20 Training accuracy: 66.20% Validation accuracy: 61.00%
Epoch: 21 Training accuracy: 52.40% Validation accuracy: 51.00%
Epoch: 22 Training accuracy: 56.65% Validation accuracy: 54.75%
Epoch: 23 Training accuracy: 62.55% Validation accuracy: 58.50%
Epoch: 24 Training accuracy: 65.75% Validation accuracy: 61.25%
Epoch: 25 Training accuracy: 57.65% Validation accuracy: 55.75%
Epoch: 26 Training accuracy: 65.65% Validation accuracy: 61.25%
Epoch: 27 Training accuracy: 57.50% Validation accuracy: 56.00%
Epoch: 28 Training accuracy: 74.70% Validation accuracy: 70.00%
Epoch: 29 Training accuracy: 69.10% Validation accuracy: 64.25%
Epoch: 30 Training accuracy: 77.65% Validation accuracy: 71.50%
Epoch: 31 Training accuracy: 73.40% Validation accuracy: 68.25%
Epoch: 32 Training accuracy: 67.25% Validation accuracy: 62.75%
Epoch: 33 Training accuracy: 71.60% Validation accuracy: 65.50%
Epoch: 34 Training accuracy: 75.10% Validation accuracy: 69.00%
Epoch: 35 Training accuracy: 80.40% Validation accuracy: 74.00%
Epoch: 36 Training accuracy: 54.45% Validation accuracy: 52.50%
Epoch: 37 Training accuracy: 74.35% Validation accuracy: 68.00%
Epoch: 38 Training accuracy: 73.80% Validation accuracy: 68.50%
Epoch: 39 Training accuracy: 69.55% Validation accuracy: 64.25%
Epoch: 40 Training accuracy: 62.80% Validation accuracy: 58.50%
Epoch: 41 Training accuracy: 66.05% Validation accuracy: 60.75%
Epoch: 42 Training accuracy: 69.70% Validation accuracy: 64.25%
Epoch: 43 Training accuracy: 68.15% Validation accuracy: 63.25%
Epoch: 44 Training accuracy: 63.90% Validation accuracy: 59.50%
Epoch: 45 Training accuracy: 73.90% Validation accuracy: 68.25%
Epoch: 46 Training accuracy: 74.85% Validation accuracy: 68.25%
Epoch: 47 Training accuracy: 75.45% Validation accuracy: 69.75%
Epoch: 48 Training accuracy: 68.30% Validation accuracy: 61.75%
Epoch: 49 Training accuracy: 75.45% Validation accuracy: 70.25%
Epoch: 50 Training accuracy: 84.85% Validation accuracy: 79.00%
Epoch: 51 Training accuracy: 63.35% Validation accuracy: 57.25%
Epoch: 52 Training accuracy: 82.30% Validation accuracy: 74.50%
Epoch: 53 Training accuracy: 84.00% Validation accuracy: 79.50%
Epoch: 54 Training accuracy: 79.40% Validation accuracy: 75.00%
Epoch: 55 Training accuracy: 72.25% Validation accuracy: 67.50%
Epoch: 56 Training accuracy: 81.65% Validation accuracy: 76.25%
Epoch: 57 Training accuracy: 85.95% Validation accuracy: 80.50%
Epoch: 58 Training accuracy: 76.35% Validation accuracy: 70.50%
Epoch: 59 Training accuracy: 62.50% Validation accuracy: 56.25%
Epoch: 60 Training accuracy: 71.70% Validation accuracy: 67.00%
Epoch: 61 Training accuracy: 72.10% Validation accuracy: 67.25%
Epoch: 62 Training accuracy: 77.50% Validation accuracy: 72.00%
Epoch: 63 Training accuracy: 74.65% Validation accuracy: 70.00%
Epoch: 64 Training accuracy: 67.20% Validation accuracy: 61.50%
Epoch: 65 Training accuracy: 79.30% Validation accuracy: 72.25%
Epoch: 66 Training accuracy: 88.30% Validation accuracy: 82.00%
Epoch: 67 Training accuracy: 80.70% Validation accuracy: 75.25%
Epoch: 68 Training accuracy: 82.85% Validation accuracy: 77.50%
Epoch: 69 Training accuracy: 81.30% Validation accuracy: 77.50%
Epoch: 70 Training accuracy: 85.95% Validation accuracy: 81.25%
Epoch: 71 Training accuracy: 88.55% Validation accuracy: 83.25%
Epoch: 72 Training accuracy: 78.00% Validation accuracy: 72.00%
Epoch: 73 Training accuracy: 71.40% Validation accuracy: 64.50%
Epoch: 74 Training accuracy: 87.60% Validation accuracy: 79.75%
Epoch: 75 Training accuracy: 92.30% Validation accuracy: 86.00%
Epoch: 76 Training accuracy: 70.65% Validation accuracy: 65.00%
Epoch: 77 Training accuracy: 78.15% Validation accuracy: 70.75%
Epoch: 78 Training accuracy: 79.80% Validation accuracy: 73.50%


Perceptron classifier
File
Tokenize
Initialize optimizer
Start optimizer
Stop optimizer
Epoch: 83 Training accuracy: 90.75% Validation accuracy: 82.25%
Epoch: 84 Training accuracy: 90.80% Validation accuracy: 82.75%
Epoch: 85 Training accuracy: 88.50% Validation accuracy: 79.75%
Epoch: 86 Training accuracy: 86.00% Validation accuracy: 77.75%
Epoch: 87 Training accuracy: 79.90% Validation accuracy: 73.75%
Epoch: 88 Training accuracy: 82.95% Validation accuracy: 75.75%
Epoch: 89 Training accuracy: 83.00% Validation accuracy: 75.00%
Epoch: 90 Training accuracy: 87.25% Validation accuracy: 79.25%
Epoch: 91 Training accuracy: 81.75% Validation accuracy: 73.25%
Epoch: 92 Training accuracy: 90.75% Validation accuracy: 84.50%
Epoch: 93 Training accuracy: 83.85% Validation accuracy: 76.00%
Epoch: 94 Training accuracy: 84.55% Validation accuracy: 75.25%
Epoch: 95 Training accuracy: 80.10% Validation accuracy: 71.50%
Epoch: 96 Training accuracy: 91.55% Validation accuracy: 85.00%
Epoch: 97 Training accuracy: 93.45% Validation accuracy: 86.00%
Epoch: 98 Training accuracy: 89.20% Validation accuracy: 80.50%
Epoch: 99 Training accuracy: 95.90% Validation accuracy: 90.00%
Epoch: 100 Training accuracy: 92.60% Validation accuracy: 86.00%
Epoch: 101 Training accuracy: 89.60% Validation accuracy: 80.75%
Epoch: 102 Training accuracy: 95.75% Validation accuracy: 89.50%
Epoch: 103 Training accuracy: 78.95% Validation accuracy: 71.75%
Epoch: 104 Training accuracy: 87.75% Validation accuracy: 78.50%
Epoch: 105 Training accuracy: 94.85% Validation accuracy: 88.25%
Epoch: 106 Training accuracy: 93.20% Validation accuracy: 85.25%
Epoch: 107 Training accuracy: 89.65% Validation accuracy: 80.50%
Epoch: 108 Training accuracy: 84.20% Validation accuracy: 74.75%
Epoch: 109 Training accuracy: 94.45% Validation accuracy: 86.75%
Epoch: 110 Training accuracy: 90.75% Validation accuracy: 80.50%
Epoch: 111 Training accuracy: 92.95% Validation accuracy: 85.00%
Epoch: 112 Training accuracy: 94.85% Validation accuracy: 88.25%
Epoch: 113 Training accuracy: 87.35% Validation accuracy: 77.75%
Epoch: 114 Training accuracy: 78.15% Validation accuracy: 69.25%
Epoch: 115 Training accuracy: 95.85% Validation accuracy: 89.50%
Epoch: 116 Training accuracy: 86.00% Validation accuracy: 75.50%
Epoch: 117 Training accuracy: 93.50% Validation accuracy: 85.00%
Epoch: 118 Training accuracy: 97.10% Validation accuracy: 90.75%
Epoch: 119 Training accuracy: 90.50% Validation accuracy: 79.75%
Epoch: 120 Training accuracy: 95.45% Validation accuracy: 88.50%
Epoch: 121 Training accuracy: 91.40% Validation accuracy: 80.00%
Epoch: 122 Training accuracy: 91.35% Validation accuracy: 81.00%
Epoch: 123 Training accuracy: 95.65% Validation accuracy: 89.25%
Epoch: 124 Training accuracy: 93.00% Validation accuracy: 84.25%
Epoch: 125 Training accuracy: 97.05% Validation accuracy: 90.50%
Epoch: 126 Training accuracy: 94.50% Validation accuracy: 86.25%
Epoch: 127 Training accuracy: 92.85% Validation accuracy: 83.25%
Epoch: 128 Training accuracy: 92.10% Validation accuracy: 81.00%
Epoch: 129 Training accuracy: 97.40% Validation accuracy: 90.50%
Epoch: 130 Training accuracy: 96.80% Validation accuracy: 90.50%
Epoch: 131 Training accuracy: 97.30% Validation accuracy: 90.75%
Epoch: 132 Training accuracy: 97.15% Validation accuracy: 90.25%
Epoch: 133 Training accuracy: 91.05% Validation accuracy: 81.00%
Epoch: 134 Training accuracy: 84.15% Validation accuracy: 75.00%
Epoch: 135 Training accuracy: 96.85% Validation accuracy: 90.25%
Epoch: 136 Training accuracy: 96.55% Validation accuracy: 90.50%
Epoch: 137 Training accuracy: 98.20% Validation accuracy: 92.00%
Epoch: 138 Training accuracy: 88.55% Validation accuracy: 78.75%
Epoch: 139 Training accuracy: 91.00% Validation accuracy: 82.00%
Epoch: 140 Training accuracy: 98.55% Validation accuracy: 91.50%
Epoch: 141 Training accuracy: 96.65% Validation accuracy: 89.75%
Epoch: 142 Training accuracy: 94.90% Validation accuracy: 86.75%
Epoch: 143 Training accuracy: 97.90% Validation accuracy: 91.00%
Epoch: 144 Training accuracy: 93.05% Validation accuracy: 83.50%
Epoch: 145 Training accuracy: 96.10% Validation accuracy: 89.25%
Epoch: 146 Training accuracy: 95.20% Validation accuracy: 87.75%
Epoch: 147 Training accuracy: 94.80% Validation accuracy: 86.00%
Epoch: 148 Training accuracy: 92.05% Validation accuracy: 82.50%
Epoch: 149 Training accuracy: 95.35% Validation accuracy: 88.25%
Epoch: 150 Training accuracy: 96.85% Validation accuracy: 90.25%
Epoch: 151 Training accuracy: 96.80% Validation accuracy: 90.50%
Epoch: 152 Training accuracy: 94.65% Validation accuracy: 87.50%
Epoch: 153 Training accuracy: 99.10% Validation accuracy: 93.25%
Epoch: 154 Training accuracy: 74.10% Validation accuracy: 68.00%
Epoch: 155 Training accuracy: 97.35% Validation accuracy: 91.25%
Epoch: 156 Training accuracy: 92.65% Validation accuracy: 82.50%
Epoch: 157 Training accuracy: 98.20% Validation accuracy: 92.00%
Epoch: 158 Training accuracy: 86.70% Validation accuracy: 77.00%
Epoch: 159 Training accuracy: 96.35% Validation accuracy: 90.50%
Epoch: 160 Training accuracy: 95.90% Validation accuracy: 89.75%
Epoch: 161 Training accuracy: 98.45% Validation accuracy: 92.00%
Epoch: 162 Training accuracy: 98.95% Validation accuracy: 92.50%



In the figures above, the optimiser is started using the ‘Start optimizer’ button and the training and validation accuracies of each epoch is printed.



In the figure above, I found that the best classifier has a training accuracy of 100%, validation accuracy of 93.5% and test accuracy of 94.5%

One correctly classified sentence and one incorrectly classified sentence are shown above.

The top 10 most positive words in descending order are comfortable, best, good, excellent, thank, nice, great, easy, very, happy

The top 10 most negative words in descending order are worst, told, never, old, no, customer, bad, not, poor, rude