Assignment 1,
Tagger and Classifiers

Fredrik Sitje
950314-8232

February 2024

Problem 1.1, Part-of-speech (POS) tagging

In this section the the part-of-speech (POS) tagging is investigated, by implementing and evaluating a unigram tagger in `C#`, and then evaluating an existing `Python` implementation of a perceptron tagger. The data in use is the `BrownCorpus.txt` aswell as a text file `BrownToUniversalMapping.txt` to handle conversion from brown tags to universal tags.

Task (a): Unigram tagger in `C#`

The unigram tagger is an algorithm based on statistics of a chosen ground truth that assigns tags to words it has statistics on. The way it does it is fairly simple but with surprisingly high accuracy. In this case the unigram tagger gets quasi-trained on a text file called the Brown Corpus, which has over 1 million words with a tag on each word. A tag tells what class a word belongs to e.g. noun, verb, pronoun, etc.. The unigram tagger in this case stores each word with its assigned tag and keeps statistics about the frequency of tags per word. This is essential, because there are words with more than one tag e.g. the word *bear*, which can be verb or noun depending on the context. The unigram tagger will then simply tag an instance of a word with the tag with the highest count. This will necessarily lead to false tags since it does not take context into account as already mentioned. Further, the unigram tagger is not able to set tags for words that are not stored in its database/the Brown corpus. Nevertheless here are some statistics of the unigram tagger applied to a training set (80% of the Brown corpus):

Table 1: Generated statistics on a training set of 962957 words (tokens) $\sim$ 80% of the `BrownCorpus.txt`. Notice that the Brown tag set ($\sim$ 500 tags) is reduced to twelve universal tags. The mapping instructions are stored in `BrownToUniversalMapping.txt`.

Universal tag	**Count**	**Relative share**
NOUN	241680	25.10%
VERB	150590	15.64%
ADP	126422	13.13%
DET	117090	12.16%
.	101152	10.50%
ADJ	73936	7.68%
ADV	45994	4.78%
PRON	35610	3.70%
CONJ	32195	3.34%
PRT	23349	2.42%
NUM	13886	1.44%
X	1053	0.11%
Total	962957	100%

Table 2: Statistics were generated from a training set based on $\sim$ 80% of the `BrownCorpus.txt`. Words by number of associated POS tags. The most of the words only have one tag, while there is one word with six tags. It's the word *down*.

# tags	**# words**	**Relative share**
1	42464	93.40%
2	2780	6.11%
3	168	0.41%
4	29	0.06%
5	3	0.01%
6	1	0.00%
Total	45445	99.99%

As one can see from the tables 1 and 2, over half the words in the training set are either nouns, verbs or adpositions (a cover term for prepositions and postpositions). Also the majority of the in total $\sim$ 45k words only have one tag, while there is one word (*down*) with six tags. By "training" the unigram tagger on these statistics and applying it on the remaining 20% of the Brown corpus words, it is possible to calculate a measure of accuracy by comparing the true tags versus the tags set by the unigram tagger. The unigram tagger achieves an accuracy of 90.98%.

Implementation in C#

These results were obtained by using a predefined solution but with missing classes. No GUI building was needed. Firstly the `BrownCorpus.txt` was read in using a previously provided function and thereby stored in `completeDataSet`. With the same method the `BrownToUniversalMapping.txt` was read in except for the difference that the function should split the string at the tabulator instead of the space. Then each pair of tags (Brown tag and corresponding universal tag) were matched and stored in an object of the class `TagConversionPair` which stored the `oldTag` and `newTag` as can be seen in the following code snippet.

```
List<string> oldTagNewTag = line.Split(new char[] { '\t' }, StringSplitOptions.
    RemoveEmptyEntries).ToList();
TagConversionPair tagConversionPair = new TagConversionPair();
if (oldTagNewTag.Count == 2) // Needed in order to ignore the very last line that
    just contains "_.".
{
    tagConversionPair.OldTag = oldTagNewTag[0].Trim();
    tagConversionPair.NewTag = oldTagNewTag[1].Trim();
}
completeTagConversionData.TagConversionList.Add(tagConversionPair);
```

After this step the whole information about how to convert the from Brown tag to universal tag was stored in `completeTagConversionData` of class `ConversionInstructions`. The tag conversion instructions were then used in the method `ConvertPOSTags` (a method defined in the `POSDataSet` class) to convert the tags.

```
    public void ConvertPOSTags(ConversionInstructions conversionInstructions)
{
    foreach (Sentence sentence in sentenceList)
    {
        foreach (TokenData tokenData in sentence.TokenDataList)
        {
            TagConversionPair conversionPair = conversionInstructions.
    TagConversionList.Find(pair => pair.OldTag.Equals(tokenData.Token.POSTag,
    StringComparison.OrdinalIgnoreCase));

            // If a conversion is found, update the token's POS tag
            if (conversionPair != null)
            {
                tokenData.Token.POSTag = conversionPair.NewTag;
            }
        }
    }
}
```

Then by using `.split(completeDataSet, splitFraction)` the `trainigDataSet` and `testDataSet` were created.

```
    private void splitDataSetButton_Click(object sender, EventArgs e)
{
    double splitFraction;
    Boolean splitFractionOK = double.TryParse(splitFractionTextBox.Text, out
    splitFraction);
    if (splitFractionOK && splitFraction > 0 && splitFraction <= 1)
    {

        var (trainingData, testData) = POSDataSet.Split(completeDataSet,
    splitFraction);

        // Activate buttons as before
        generateStatisticsButton.Enabled = true;
        generateUnigramTaggerButton.Enabled = true;

        trainingDataSet = trainingData;
        testDataSet = testData;

        resultsListBox.Items.Add("The complete data set was successfully split
    80/20.");
    }
    else
    {
        resultsListBox.Items.Add("Incorrectly specified split fraction");
    }
}
```

Then I initialised two dictionaries `posTagCounts` a dictionary storing a string as key and an integer as value and `wordToTags` a dictionary storing a string as key and a list of strings as value. With these it is possible to store information about how many times a POS tag is used and how many tags are used on a word. This data can then be displayed as neat statistics, as can be seen in tables 1 and 2.

```
Dictionary<string, int> posTagCounts = new Dictionary<string, int>();
Dictionary<string, List<string>> wordToTags = new Dictionary<string, List<string
    >>();
int totalCount = 0;
    foreach (Sentence sentence in trainingDataSet.Sentences)
{
    totalCount += sentence.TokenDataList.Count;
    foreach (TokenData tokenData in sentence.TokenDataList)
    {
        string word = tokenData.Token.Spelling.ToLower(); // Normalize word to
    lowercase
        string posTag = tokenData.Token.POSTag;

        // Update posTagCounts
        if (posTagCounts.ContainsKey(posTag))
        {
            posTagCounts[posTag]++;
        }
        else
        {
            posTagCounts.Add(posTag, 1);
        }

        // Update wordToTags
        if (wordToTags.ContainsKey(word))
        {
            if (!wordToTags[word].Contains(posTag))
            {
                wordToTags[word].Add(posTag);
            }
        }
        else
        {
            wordToTags[word] = new List<string> { posTag };
        }
    }
}
```

Then with the class `UnigramTagger` the Unigram tagger was generated and trained by the method `Train()`, which takes all instances of a word/token stores it in a dictionary with corresponding POS tag and its count. Then it sorts them by count, so that the highest count of POS tag will be represented on each instance of the corresponding word/token. The unigram tagger could was then ready to use by the method `Tag()`. The methods are shown below.

```
public class UnigramTagger : POSTagger
{
    private Dictionary<string, string> mostFrequentTags;

    public UnigramTagger()
    {
        mostFrequentTags = new Dictionary<string, string>();
    }

    public void Train(POSDataSet trainingDataSet)
    {
        var tokenTags = new Dictionary<string, Dictionary<string, int>>();

        foreach (var sentence in trainingDataSet.Sentences)
        {
            foreach (var tokenData in sentence.TokenDataList)
            {
                if (!tokenTags.ContainsKey(tokenData.Token.Spelling))
                    tokenTags[tokenData.Token.Spelling] = new Dictionary<string,
    int>();

                if (!tokenTags[tokenData.Token.Spelling].ContainsKey(tokenData.
    Token.POSTag))
                    tokenTags[tokenData.Token.Spelling][tokenData.Token.POSTag] =
    0;

```

```
                tokenTags[tokenData.Token.Spelling][tokenData.Token.POSTag]++;
            }
        }

        foreach (var token in tokenTags)
        {
            var mostFrequentTag = token.Value.OrderByDescending(tag => tag.Value).
    First();
            mostFrequentTags[token.Key] = mostFrequentTag.Key;
        }
    }

    public override List<string> Tag(Sentence sentence)
    {
        List<string> tags = new List<string>();

        foreach (TokenData tokenData in sentence.TokenDataList)
        {
            string spelling = tokenData.Token.Spelling;
            string tag = mostFrequentTags.ContainsKey(spelling) ? mostFrequentTags[
    spelling] : "UNK";
            tags.Add(tag);
        }
        return tags;
    }
}
```

So in the end the `UnigramTagger` is used on the `testDataSet` and the output is stored in a list of strings to compare with the real tags of the test data set. If the `UnigramTagger` predicted correctly a counter increases by one. In the end the total correct predictions is divided by the total POS tags in `testDataSet` which corresponds to the accuracy. As already stated, the unigram tagger achieved an accuracy of 90.98%.

Task (b): NLTK-perceptron tagger in `Python`

The accuracy obtained from the perceptron tagger after mapping the inferred tags to the universal POS tag set is 90.42%.

Problem 1.2, Perceptron text classifier

After training an perceptron text classifier on a training set for over 200 epochs the highest validation accuracy was scored on epoch 197 as can be seen in figure 1. The learning rate η was set to 0.05, and the accuracy reached by the best classifier can be seen in table 3. In figure 1 one finds

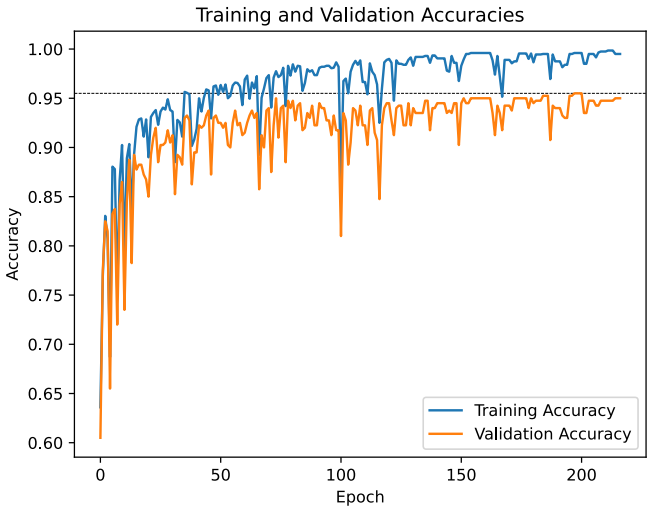

Figure 1: One can see the increasing accuracy tendency of the perceptron classification tested on the training and validation set. For each epoch the perceptron was trained on 100 randomly chosen airline reviews of the training set. The learning rate η was set to 0.05. The accuracy after 197 epochs is 0.996 for the training set and 0.955 for the validation set.

that the training of the perceptron work as expected. The blue line shows the accuracy of the perceptron on the training set and the orange line the accuracy on the validation set. It is typical that the accuracy is higher on the training set than the validation set, because the perceptron is trained how to classify based on the training set. The validation accuracy is used as an indicator of good performing classifiers where the highest scores are taken as models to be tested on the test set. This was done using the classifier from epoch 197 which scored an accuracy of 0.925, see table 3.

Table 3: The best classifier is the one with the highest validation accuracy which appeared after 197 epochs and has an accuracy of 0.925 on the test data set.

Data Set	**Accuracy**
Training	0.996
Validation	0.955
Test	0.925

Now let's look at two examples of correctly classified reviews.

Hong Kong to London via Beijing. Worst airline I have ever tried, from their online services to the flight quality. Their website is a joke, it keeps timing out and it doesn't allow you to checkin for flights that have intermediate stops. I have complained to one of their staff at the airport and they admitted that they have issues with their online check-in. The plane was also a joke, especially the one used to fly from Beijing to London. I'm 188cm tall, I didn't have enough legroom and my knee were pushed against the sit in front of me for 11 hours! The legroom was less than in a cheap low cost airline (e.g. Ryanair), how can that happen in an international flight where you need to be sit for more than ten hours! Let's ignore the cleanliness and comfort, but a minimum amount of legroom should always be provided. The flight entertainment was also embarrassing, straight out of the nineties, the resolution of the display was probably the same as one of the first smartphone ever produced, it was painful to watch, and the controllers of the entertainment system were unintuitive and most of the buttons to press were unresponsive. The meal quality was really poor, food was just a little warm, they gave us two options for the meals and they ran out of the first option immediately, I think they managed to serve maybe ten persons and then they ran out of it, speechless! To

conclude our trip properly they damaged one of our luggage, we found the luggage completely open running around on the belt at the luggage collection, we obviously filled in an official complaint as part of the content was broken/lost. I think, no matter how cheap their flights are, we will never book a flight with them, worst experience ever!,

and the second one,

Inhumane seats. I fly constantly, maybe 50 flights a year. These are the worst seats I've ever "fit" in. I weigh about 210 and am 5' 11". I have shortish legs for my height. Still I can point my knees forward as they press into the seat in front of me. Every move the person makes in front of me, I feel. Now. I flew on points. So even with my Air Canada status they shoved me to the back of the plane (with my wife). They should be ashamed. My wife actually felt claustrophobic. Air Canada. Anything for a buck.

For a human both classifications seem obvious. Now lets look at two examples where the perceptron text classifier did not label the review correctly.

On October 8th 2022, we were suppose to fly with Air Canada from Jakarta to Hong Kong with Cathay Pacific then contacting flight from Hong Kong to Vancouver with Air Canada. When we went to check-in counter at Jakarta we were informed by Cathay that Air Canada has changed out flight we will have to wait more then 24 hours at Hong Kong Airport in order to fly back home and in Hong Kong they will put us in quarantine for 3 days. Cathay tried for 3 hours to contact Air Canada but they never respond there call, finally after waiting for 3 hours at Jakarta, Cathay changed our flight for next day so that we don't have to wait for 24 hour and we can still fly out of Hong Kong with our original flight Air Canada on 9th October. The next day again we went back to Jakarta Airport to fly with Cathay Pacific to Hong Kong then with contacting flight with Air Canada again we found out that Air Canada has cancelled our reservation. We ended up to buy another 2 tickets from Jakarta to Vancouver for $1400 dollars. Never again fly with Air Canada incompetent airline which is the worst.

and the second one,

Delhi-Barcelona. I could not check-in online so called up the airlines to find my flight was rescheduled. My connecting flight had been rescheduled so that my transit took 24 hrs and they wouldn't provide any sort of accommodation. Moscow airport contains narrow corridors with duty free shops food must be paid with Russian money. There was free Internet but only in a few places with low wifi signal. Plane food was bland and mediocre. Average service.

While the first of them has a clear statement in the end which indicates clearly that it's a negative review, the second one is not very positive but not extremely negative it feels more like an explanation of an average airline.

It might help too have a look at the biggest drivers of positive and negative labels as can be seen in tables 4 and 5. While the words shown in the table 4 which shows the tokens (words) with

Table 4: Top 10 positive words

Word	Index	Value
good	5091	4.044
excellent	4392	3.976
great	5129	3.745
comfortable	2941	3.213
best	2138	2.967
thank	10171	2.894
friendly	4904	2.881
nice	7194	2.859
helpful	5369	2.811
always	1496	2.648

Table 5: Top 10 negative words

Word	Index	Value
not	7273	-3.450
"	7	-3.091
told	10332	-2.558
no	7220	-2.479
never	7174	-2.310
they	10208	-2.210
rude	8922	-2.089
customer	3414	-2.061
because	2063	-2.025
at	1783	-2.024

the highest contributions for positive labelling, are what one usually expects, some of the words in table 5 are surprising. The words *they*, *told* and the apostrophe " seem unexpected to contribute to

negative labels while the rest make sense when thinking about the context of the negative reviews.

Implementation in C#

To implement the perceptron text classifier in C# the method of reading in data and tokenization of the data was copied from the POSTagger in Problem 1.1. The tokenizer had to be changed quiet a lot e.g. to take into account abbreviations, decimal numbers, and many other cases. That can be found in the class Tokenizer. Form these tokens I created a vocabulary based on the tokens of the training set. This was done using the class Vocabulary and the

```
public class Vocabulary
{
    public Dictionary<string, int> WordIndex { get; set; }
    private int Count = 0;
    private int IndexCount = 0;

    public Vocabulary()
    {
        WordIndex = new Dictionary<string, int>();
    }

    public void AddWord(string Word)
    {
        if (!WordIndex.ContainsKey(Word))
        {
            WordIndex.Add(Word, IndexCount);
            Count++;
            IndexCount++;
        }
    }

    public string GetString(int Index)
    {
        foreach (var pair in WordIndex)
        {
            if (pair.Value == Index)
            {return pair.Key;}
        }
        return null;
    }

    public int GetIndex(string Word)
    {
        if (WordIndex.ContainsKey(Word))
        {return WordIndex[Word];}
        else
        {return -1;}
    }

    public int GetCount()
    {return Count;}
    public int GetLastIndex()
    {return IndexCount;}
}
```

Then using the vocabulary it is possible to index the the reviews i.e. converting reviews to vectors containing indexes. Here an example of indexing the training set on clicking the button in the GUI.

```
    int trainingCounter = 0;
foreach (Sentence sentence in tokenizedTrainingSet)
{
    List<int> sentenceAsVocabularyIndexes = new List<int>();
    foreach (TokenData tokenData in sentence.TokenDataList)
    {
        string word = tokenData.Token.Spelling;

        int vocabularyIndex = vocabulary.GetIndex(word);
        tokenData.SetTheIndex(vocabularyIndex);
        sentenceAsVocabularyIndexes.Add(vocabularyIndex);
    }
    trainingSet.ItemList[trainingCounter].ReviewAsVocabularyIndexes =
    sentenceAsVocabularyIndexes;
    trainingCounter++;
}
```

This is done to increase the efficiency of finding the right weights of the perceptron to train. There are as many weights created as there are words in the vocabulary and they are initialised with random numbers between 0 and 1. So when a review of the training set is fed in as a list of index to train the weights it looks up the weights by index and trains them in steps of ± 0.05 depending on if the target is met or not. The weights were then trained as explained before. One crucial aspect is the fact that threading was implemented for the optimisation, so that there could be presented updates regularly in the GUI. This was accomplished using the following

```
private volatile bool shouldStop = false;

private void ShowProgress(string progressInformation)
{
    progressListBox.Items.Add(progressInformation);
}

private void startOptimizerButton_Click(object sender, EventArgs e)
{
    startOptimizerButton.Enabled = false;
    progressListBox.Items.Clear();
    progressListBox.Items.Add("Starting");

    computationThread = new Thread(new ThreadStart(() => ComputationLoop()));
    computationThread.Start();

    stopOptimizerButton.Enabled = true; // Enable the stop button
}

private void ComputationLoop()
{
    int firstSave = 1;
    int epoch = 0;
    while (!shouldStop) // Continue until stop signal is received
    {
        (...computation intensive code...)
    }
    ThreadSafeHandleDone();
}

private void ThreadSafeHandleDone()
    {
        if (InvokeRequired)
        {BeginInvoke(new MethodInvoker(() => { HandleDone(); }));}
        else
        {HandleDone();}
    }

private void ThreadSafeShowProgress( string progressInformation)
    {
        if (InvokeRequired)
        {BeginInvoke(new MethodInvoker(() => { ShowProgress(progressInformation);
    }));}
        else
        {ShowProgress(progressInformation);}
    }
private void HandleDone()
    {
        startOptimizerButton.Enabled = true; // Enable start button
        stopOptimizerButton.Enabled = false; // Disable stop button
    }

private void stopOptimizerButton_Click(object sender, EventArgs e)
    {
        stopOptimizerButton.Enabled = false;
        shouldStop = true; // Signal stop to computation loop

        (..code to print best calssifier accuracy on test set..)

        stopOptimizerButton.Enabled = true; // A bit ugly, should wait for the
    }
}
```

This implementation makes serves the purpose of not letting the GUI freeze under heavy computation load.

When the training is done one tokenizes and vectorizes the validation and test set based on the

vocabulary and lets the perceptron classify each of the reviews. The accuracy is then calculated by dividing the amount correctly predicted labels by the total amount of labels.

Problem 1.3, Bayesian text classifier

The priori porbabilities were calculated using the formula

$$P(c_j) = \frac{\text{count}(c_j)}{m} \tag{1}$$

Which turned out to be $P(\text{label } 0) = 0.5$ and $P(\text{label } 1) = 0.5$, because there are equal amounts of label 0 as there are of label 1. The conditional probabilities of finding either of the class labels depending on one of these words *friendly, perfectly, horrible* and *poor* is given in table 6. They

Table 6: Conditional Probabilities

P(label | word)	**friendly**	**perfectly**	**horrible**	**poor**
negative	0.1023	0	1	1
positive	0.8977	1	0	0

were calculated using Bayes' rule

$$P(A|B) = \frac{P(A)\,P(B|A)}{P(B)}. \tag{2}$$

The performance of the Bayesian text classifier over the test set was measured using precision, recall, accuracy and F1.

$$\text{Precision (P)}: \quad P = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3}$$

$$\text{Recall (R)}: \quad R = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4}$$

$$\text{Accuracy (A)}: \quad A = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{5}$$

$$\text{F1 Score (F1)}: \quad F1 = \frac{2PR}{P + R} \tag{6}$$

Where the abriviations are the following TP := true positive, TN := true negative, FT := false positive and FN := false negative. The results are shown in tables 7 and 8. The accuracy obtained

Table 7: Analysis on the training set

Metric	**Value**
Precision	0.9603
Recall	0.9200
Accuracy	0.9410
F1 Score	0.9397

Table 8: Analysis on the test set

Metric	**Value**
Precision	0.9744
Recall	0.9268
Accuracy	0.9600
F1 Score	0.9500

from the Bayesian text classifier trained on the airline reviews gave an accuracy of 0.93 on the corresponding test set. This 0.05 higher than the perceptron achieved over the same data set.

Implementation in `C#`

For the implementation the tokenizer of Problem 1.2 was used. Then a "bag of words" was created, which would store each word (token) appearing in the training set and how often it was used in a negative (0) or positive (1) context. Then a `NaiveBayesianTextClassifier` class was generated which would classify each review according to this calculation

```
1 public int Classify(Sentence tokenizedReview, Dictionary<string,TokenData>
      bagOfWords, double class0Denominator, double class1Denominator, double
      class0LabelProb, double class1LabelProb)
2 {
3     // Implement the caluclations to find the class Label.
4     double class0LabelProbability = Math.Log(class0LabelProb);
5     double class1LabelProbability = Math.Log(class1LabelProb);
6     foreach (TokenData tokenData in tokenizedReview.TokenDataList)
7     {
8         if (bagOfWords.ContainsKey(tokenData.Token.Spelling))
```

```
        {
            double class0LabelCounts = bagOfWords[tokenData.Token.Spelling].
    Class0Count;
            double class0TokenProb = (class0LabelCounts + 1) / class0Denominator;
            class0LabelProbability += Math.Log(class0TokenProb);

            double class1LabelCounts = bagOfWords[tokenData.Token.Spelling].
    Class1Count;
            double class1TokenProb = (class1LabelCounts + 1) / class1Denominator;
            class1LabelProbability += Math.Log(class1TokenProb);
        }
    }
    if (class0LabelProbability > class1LabelProbability)
    {return 0;}
    else
    {return 1;}
}
```

Which essentially is an translated `C#` version of this formula

$$\hat{c} = \text{argmax}_{j \in \{1,2,\ldots,k\}} \left(\log \hat{P}(c_j) + \sum_{i=1}^{v} \log \hat{P}(w_i|c_j) \right). \tag{7}$$

In the same spirit the precision, recall, accuracy and F1 score equations (3,4,5,6) were implemented.

Problem 1.4, Autocompletion with n-grams

My implementation of the autocompletion with n-grams is based on three n-grams which in fact are alphabetically ordered dictionaries that contain the n-gram as a string as key (identifier) while the value is an object of class `NGram`. The `NGram` object holds three metrics, the whole n-gram string, a list of strings which is made up of the split n-gram string, a frequency and a frequency per one million. In addition logical conditions lead to the next recommended word.

```
public class NGram
{
    public string Identifier { get; set; }

    public List<string> TokenList { get; set; }

    public double FrequencyPerMillionInstances { get; set; }

    public int Frequency { get; set; }

    public NGram(string identifier)
    {
        Identifier = identifier.Trim();

        TokenList = identifier.Split(new char[] { ' ' }, StringSplitOptions.
    RemoveEmptyEntries).ToList();

        for (int i = 0; i < TokenList.Count; i++)
        {
            TokenList[i] = TokenList[i].Trim();
        }
    }
}
```

Then I use the same methods as in one of the earlier problems to read in the text files, and tokenize them with the same tokenizer as usual but modified to suit lists of `strings` instead of lists of `TokenData` as before. So when the data is tokenized is have three separate methods (buttons) to create the unigram, bigram and trigram. Essentially they go over the tokenized data and create the different strings of the specified length. Then after they have been created I create a dictionary as explained above, and add the key and value pair to the dictionary if its not already in it. If it's already present the count of frequency is increased by one. Here the implementation of the unigramDictionary

```
private void GenerateUnigramsButton_Click(object sender, EventArgs e)
{
    Dictionary<string, NGram> UnigramDictionary = new Dictionary<string, NGram>();
    foreach (Sentence review in tokenizedCompleteDataSet)
    {
        foreach (string token in review.TokenDataList)
        {
            NGram unigram = new NGram(token);

            if (UnigramDictionary.ContainsKey(token))
            {
                UnigramDictionary[token].Frequency++;
            }
            else
            {
                unigram.Frequency++;
                UnigramDictionary.Add(token, unigram);

            }
        }
    }

    foreach (KeyValuePair<string, NGram> entry in UnigramDictionary)
    {
        NGram unigram = entry.Value;
        int frequency = unigram.Frequency;
        unigram.FrequencyPerMillionInstances = (double)frequency / (1000000);
    }

    UnigramDictionary = UnigramDictionary.OrderBy(x => x.Key).ToDictionary(x => x.
    Key, x => x.Value);
    unigramDictionary = UnigramDictionary;
    progressListBox.Items.Add($"There are {unigramDictionary.Count} unique unigrams
     in the alphabetically sorted dictionary.");
```

```
33 }
```

Notice that the dictionary gets alphabetically ordered in the end. The user can then interact with the GUI by initialising the program using the buttons and then start entering words into the text box. The input is then internally converted to a string. Then there is a case distinction where either the string is made up of two words or more, or only one. In case there is more than two words, only the two last words of the string are considered. What happens there after is for only one word in the string it checks first the `trigramDictionary` for trigrams that start with a full stop (.) and has the input word as the second word. It displays then the third word of the trigram with the highest `FrequencyPerMillionInstances`, if it does not find any predictions it will do the same in the bigram dictionary. If there are exactly two words it checks the trigram which has the first word in the first position and the second in the second position. Then it displays the third word, but if there are no cases, it does the same with the bigram but only with the last word (as described above). In any case it's always the case that the highest `FrequencyPerMillionInstances` is shown. This continues on for as long as one wishes, but it must be said that the auto completion becomes quiet slow with big data sets (mine around 35Mb), but its still usable.