# Explain Code for C# implementation

```
private List<string[]> tagMappings = null;
private UnigramTagger unigramTagger = null;
```

- tagMappings is used for mapping POS tag from Brown Corpus to Universal POS tags
- unigramTagger is a class used for the creation and implementation of Unigram Tagger

```
private void loadTagConversionDataToolStripMenuItem_Click(object sender, EventArgs e)
{
    /* ...
    tagMappings = new List<string[]>();

    using (OpenFileDialog openFileDialog = new OpenFileDialog())
    {
        openFileDialog.Filter = TEXT_FILE_FILTER;
        if (openFileDialog.ShowDialog() == DialogResult.OK)
        {
            StreamReader fileReader = new StreamReader(openFileDialog.FileName);
            while (!fileReader.EndOfStream)
            {
                string line = fileReader.ReadLine();
                // Process data here ...
                if (line!="")//split into each line of text which is the mapping
                {
                    string[] parts = line.Split(new char[] { '\t', ' ' }, StringSplitOptions.RemoveEmptyEntries);
                    if (parts.Length == 2)
                    {
                        tagMappings.Add(new string[] { parts[0], parts[1] });//create tagMappings of [Brown mapping , Universal mapping]
                    }
                }
            }
            fileReader.Close();
            resultsListBox.Items.Add("Loaded the Tag Conversion Data");
        }
    }

    // Keep these lines: They will activate the conversion button, provided that the
    // Brown data set has been loaded first:
    if (completeDataSet != null)
    {
        if (completeDataSet.SentenceList.Count > 0)
        {
            convertPOSTagsButton.Enabled = true;
        }
    }
}
```

This code is used to load the Brown Corpus POS tag to Universal POS tag conversion into the application. The file is divided based on the line the text is on and then split into individual Brown POS tag-and-Universal POS tag items. The item is then separated into Brown POS tag and Universal POS tag which forms a string array as element 0 and 1. Then they are added to tagMappings, creating [Brown mapping, Universal mapping]. The file is closed and the text “Loaded the Tag Conversion Data” is printed on the application.

```
private string GetMappedTag(string originalTag, List<string[]> tagMappings)
{
    foreach (string[] mapping in tagMappings)
    {
        if (mapping[0] == originalTag)//if left element of tagMapping  same as original Brown POSTag, change to Universal POSTag
            return mapping[1];
    }
    return "UNKNOWN";
}

private void convertPOSTagsButton_Click(object sender, EventArgs e)
{
    // Write code here, such that the Brown tags are mapped to the
    // Universal tags (for the complete data set), using the representation described above
    // After running this method, all the tokens should be assigned
    // one of the 12 Universal tags.
    //
    // Method call:
    // completeDataSet.ConvertPOSTags(... <suitable input, namely the tag conversion data> ...); // this you have to write ...
    POSDataSet newDataSet = new POSDataSet();

    foreach (Sentence sentence in completeDataSet.SentenceList)
    {
        Sentence newSentence = new Sentence();

        foreach (TokenData tokenData in sentence.TokenDataList)
        {
            string originalTag = tokenData.Token.POSTag;
            string newTag = GetMappedTag(originalTag, tagMappings);

            Token newToken = new Token
            {
                Spelling = tokenData.Token.Spelling,
                POSTag = newTag
            };

            newSentence.TokenDataList.Add(new TokenData(newToken));
        }

        newDataSet.SentenceList.Add(newSentence);

    }

    completeDataSet = newDataSet;

    resultsListBox.Items.Add("Converted the Brown Corpus to Universal");
    // Next, build the vocabulary, using the 12 universal tags (this method you get for free! :) )
    // NOTE: (Only) in this problem (for simplicity) the vocabulary is a simple List<TokenData> rather
    // than an instance of the Vocabulary class (which defines a Dictionary<string, Token>)
    vocabulary = GenerateVocabulary(completeDataSet);
    resultsListBox.Items.Add("Vocabulary is generated");

    // Keep this line: It will activate the split button.
    splitDataSetButton.Enabled = true;
}
```

GetMappedTag is a function that takes in the original Brown POS tag and the tagMapping as its parameters. It then searches for the matching original Brown POS tag in tagMapping such that the Universal POS tag can be returned from the function. If no such tag is found in tagMapping, return “UNKNOWN”.

convertPOSTagsButtonClick triggers when the ‘Convert POS tags (Brown -> Universal)’ button is pressed. After navigating to the token level from a list of sentences, the original Brown POS tag is converted to its Universal tag using GetMappedTag. This is all assigned to a temporary POSDataSet which will be assigned to completeDataSet afterwards. I also added the texts “Converted the Brown Corpus to Universal” and “Vocabulary is generated” to be printed on the application.

```
private void splitDataSetButton_Click(object sender, EventArgs e)
{
    // Split the data set into a training set and a test set (a validation
    // set is not needed here, since no optimization is carried out - the
    // unigram tagger is as it is - no optimization required or possible).
    // The result should be
    //
    //  trainingDataSet (containing, by default, 80% of the sentences)
    //
    //  testDataSet (contaning the remaining 20% of the sentences)
    //
    double splitFraction;
    Boolean splitFractionOK = double.TryParse(splitFractionTextBox.Text, out splitFraction);

    if (splitFractionOK && splitFraction > 0 && splitFraction <= 1)
    {
        // NOTE: The most elegant way to do this is to write a static method in the POSDataSet class,
        // such as, POSDataSet.Split(POSDataSet completeDataSet, double splitFraction).
        // One should always strive to put methods *where they naturally belong*. In this case,
        // the split method belongs with the POSDataSet. One can also, of course,
        // just write the code here (in this method), instantiating the trainingDataSet and
        // the testSet, and then just adding sentences, but the most elegant way is
        // to define a method in the POSDataSet class. You can read about static
        // methods on MSDN or StackOverflow, for example
        int totalSentences = completeDataSet.SentenceList.Count;
        int trainSize = (int)(splitFraction * totalSentences);

        trainingDataSet = new POSDataSet();
        testDataSet = new POSDataSet();

        //Note: GetRange(starting index, number of elements to include in list)
        trainingDataSet.SentenceList = completeDataSet.SentenceList.GetRange(0, trainSize);
        testDataSet.SentenceList = completeDataSet.SentenceList.GetRange(trainSize, totalSentences - trainSize);


        resultsListBox.Items.Add("Training and Test sets are generated");

        // Keep these lines: It will activate the statistics generation button and the unigram tagger generation button,
        // once the data set has been split.
        generateStatisticsButton.Enabled = true;
        generateUnigramTaggerButton.Enabled = true;
    }
    else//when the splitFraction value cannot be parsed or not within [0,1]
    {
        MessageBox.Show("Incorrectly specified split fraction", "Error", MessageBoxButtons.OK);
    }
}
```

In the splitDataSetButton_Click function, it is triggered when “Split” is clicked in the application. Variable splitFraction of type double is created and then checked if it is a valid number between 0 and 1. The total number of sentences in the Brown Corpus is deduced which allows the train size (of the first 80%) sentences to be then found. The trainingDataSet and testDataSet are instantiated and they are then assigned sentence lists from completeDataSet using the trainSize. “Training and Test sets are generated” is then printed on the application

```
private void generateStatisticsButton_Click(object sender, EventArgs e)
{
    resultsListBox.Items.Clear(); // Keep this line.
    /* Write code here for carrying out all the steps described in ...

    /* In training set:
     * 1) Count the instances of each POS tag
     * 2) Compute the fractions of each POS tag wto total
     * 3) Count fraction of words associated with 1,2,... different POS tags
     */

    Dictionary<string, int> posTagCounts = new Dictionary<string, int>();
    int totalTokenCount = 0;
    Dictionary<string, HashSet<string>> wordToPosTags = new Dictionary<string, HashSet<string>>();//Word to POS tags


    foreach (Sentence sentence in trainingDataSet.SentenceList)
    {
        foreach (TokenData tokenData in sentence.TokenDataList)
        {
            string trainToken = tokenData.Token.Spelling;
            string trainTag = tokenData.Token.POSTag;

            if (!posTagCounts.ContainsKey(trainTag))
            {
                posTagCounts[trainTag] = 0;//if POS tag not in dictionary yet
            }
            posTagCounts[trainTag]++;//else add to the POS tag's count

            totalTokenCount++;//increase the total count

            if (!wordToPosTags.ContainsKey(trainToken))
            {
                wordToPosTags[trainToken] = new HashSet<string>();//if a word does not have POS tag in dictionary yet
            }
            wordToPosTags[trainToken].Add(trainTag);//else add a POS tag to the word in the dictionary
        }
    }

    Dictionary<string, double> posTagFractions = new Dictionary<string, double>();
    foreach(var posTagCount in posTagCounts)// for each POS tag
    {
        double fraction = (double)posTagCount.Value / totalTokenCount;
        posTagFractions[posTagCount.Key] = fraction;//assign fraction to the POS tag
    }
```

```
    Dictionary<int, int> wordTagCount = new Dictionary<int, int>();

    foreach(var wordEntry in wordToPosTags)
    {
        int numTags = wordEntry.Value.Count;//number of POS tags for this word

        if(!wordTagCount.ContainsKey(numTags))
        {
            wordTagCount[numTags] = 0;
        }
        wordTagCount[numTags]++;
    }

    //Display results
    resultsListBox.Items.Add("POS Tag Counts:");
    foreach(var posTagCount in posTagCounts)
    {
        resultsListBox.Items.Add($"{posTagCount.Key}:{posTagCount.Value}");
    }

    resultsListBox.Items.Add("");
    resultsListBox.Items.Add("POS Tag Fractions:");
    foreach (var posTagFraction in posTagFractions)
    {
        resultsListBox.Items.Add($"{posTagFraction.Key}:{posTagFraction.Value}");
    }

    resultsListBox.Items.Add("");
    resultsListBox.Items.Add("Word-to-POS Tag Count:");
    foreach (var wordCount in wordTagCount)
    {
        resultsListBox.Items.Add($"Words with {wordCount.Key} different POS tags:{wordCount.Value}");
    }
}
```

The generatedStatisticsButton_Click function is triggered when the “Generate Statistics” button is pressed. Text on the application is cleared. A dictionary, posTagCounts, of <POS tag, instances> is created along with the total token count being set to zero. Another dictionary,wordToPosTags , of <word, HashSet<POS tag>> is also created. After navigating to the token level, for every token, its POS tag is checked if it is in posTagCounts, adding it to the dictionary if it is not and incrementing it if it is. The total token count is incremented. For every token, if it is not present in wordToPosTags, create a HashSet for it. Then add the tag to the HashSet.

A dictionary, posTagFractions, of <POS tag, fraction> is created to obtain the fraction of the POS tag

Another dictionary, wordTagCount, of <number of different POS tags for a word, number of words which meet this criteria> is created to obtain the number of words with different POS tags

Results for the trainingDataSet are then displayed.

```
private void generateUnigramTaggerButton_Click(object sender, EventArgs e)
{
    /* ...
    unigramTagger = new UnigramTagger(trainingDataSet.SentenceList);

    resultsListBox.Items.Add("");
    resultsListBox.Items.Add("Unigram tagger has been generated");

    runUnigramTaggerButton.Enabled = true;
}
```

An object of UnigramTagger class is created with parameters of the sentence list of trainingDataSet. The generatedUnigramTaggerButton_Click function is triggered when the “Generate Unigram Tagger”button is clicked

```
public class UnigramTagger : POSTagger
{
    //Dictionary of <word, mostFreqTag>
    public Dictionary<string, string> mostFreqTags;

    //Constructor for the UnigramTagger
    public UnigramTagger(List<Sentence> trainingSentences)
    {
        mostFreqTags = new Dictionary<string, string>();
        Train(trainingSentences);
    }

    //Train the unigram tagger by building the unigram model
    private void Train(List<Sentence> trainingSentences)
    {
        //Dictionary<word, Dictionary<POSTag, instances>>
        Dictionary<string, Dictionary<string, int>> unigramModel = new Dictionary<string, Dictionary<string, int>>();

        foreach (Sentence sentence in trainingSentences)
        {
            foreach (TokenData tokenData in sentence.TokenDataList)
            {
                string word = tokenData.Token.Spelling.ToLower();//Convert to lowercase
                string posTag = tokenData.Token.POSTag;

                //Add word into dictionary if not already in it
                if (!unigramModel.ContainsKey(word))
                {
                    unigramModel[word] = new Dictionary<string, int>();
                }

                //Add POSTag to word if not already in it
                if (!unigramModel[word].ContainsKey(posTag))
                {
                    unigramModel[word][posTag] = 0;
                }

                unigramModel[word][posTag]++;//Increment count of a posTag for a word

            }
        }

        foreach (var wordEntry in unigramModel)
        {
            string word = wordEntry.Key;
            var posTagCounts = wordEntry.Value;//Dict<string, int>

            //Get mostFreqTag by choosing the highest count
            var mostFreqTag = posTagCounts.OrderByDescending(kvp => kvp.Value).First().Key;

            //Store the result in the Dictonary for this current word
            mostFreqTags[word] = mostFreqTag;
        }

    }
```

```
    //Override the abstract Tag() method from POSTagger
    public override List<string> Tag(Sentence sentence)
    {
        List<string> predictedTags = new List<string>();

        foreach (TokenData tokenData in sentence.TokenDataList)
        {
            string word = tokenData.Token.Spelling.ToLower();//Convert word to lowercase

            //Get the predicted POS tag from the unigram model
            string predictedTag = mostFreqTags.ContainsKey(word) ? mostFreqTags[word] : "NN";

            predictedTags.Add(predictedTag);

        }
        return predictedTags; //Return list of predicted POS tags for the sentence
    }
}
```

In the UnigramTagger Class, a dictionary, mostFreqTags, of <word, mostFreqTags> is created. The UnigramTagger constructor is made of a list of sentences as its parameters, has mostFreqTags in it and the private Train function. The Train function takes in a list of sentences as its parameter. A dictionary, unigramModel, of <word, Dictionary<POSTag, instances>> is created. After navigating to the token level, a word is added to the unigramModel dictionary if not already in it. Then, the POSTag is added to a word if not already in it. The count of a POSTag for a word is then incremented.
For every word in the unigramModel, the most frequent tag is determined. If the mostFreqTags contains the word, assign the tag to predictedTag. Else assign NN to it.
The abstract Tag() method from the POSTagger class is overridden. At the token level, the predicted tag is added to a list of strings, predictedTags and returned.

```
private void runUnigramTaggerButton_Click(object sender, EventArgs e)
{
    resultsListBox.Items.Clear(); // Keep this line.

    /* Write code here for running the unigram tagger over the test set. ...
    int truePositives = 0;
    int falsePositives = 0;
    int falseNegatives = 0;

    foreach (Sentence sentence in testDataSet.SentenceList)
    {
        List<string> predictedTags = unigramTagger.Tag(sentence);

        //Compare predicted tags with the ground truth (true tags)
        for (int i = 0; i < sentence.TokenDataList.Count; i++)
        {
            string trueTag = sentence.TokenDataList[i].Token.POSTag;
            string predictedTag = predictedTags[i];//Unigram tagger prediction

            //Increment true posities, false positives, and false negatives
            if (predictedTag == trueTag)
            {
                truePositives++;
            }
            else
            {
                //If word exists in the unigram model but was predicted incorrectly
                if (unigramTagger.mostFreqTags.ContainsKey(sentence.TokenDataList[i].Token.Spelling.ToLower()))
                {
                    falsePositives++;
                }
                else
                {
                    //If word not in unigram model, missing tag
                    falseNegatives++;
                }
            }
        }
    }

    double precision = (truePositives + falsePositives > 0) ? (double)truePositives / (truePositives + falsePositives) : 0;
    double recall = (truePositives + falseNegatives > 0) ? (double)truePositives / (truePositives + falseNegatives) : 0;
    double fMeasure = (precision + recall > 0) ? 2 * (precision * recall) / (precision + recall) : 0;
    double accuracy = (truePositives + falsePositives + falseNegatives > 0) ? (double)truePositives / (truePositives + falsePositives + falseNegatives) : 0;

    resultsListBox.Items.Add($"True Positive: {truePositives}");
    resultsListBox.Items.Add($"False Positive: {falsePositives}");
    resultsListBox.Items.Add($"False Negative: {falseNegatives}");
    resultsListBox.Items.Add("");
    resultsListBox.Items.Add($"Precision: {precision}");
    resultsListBox.Items.Add($"Recall: {recall}");
    resultsListBox.Items.Add($"F-measure: {fMeasure}");
    resultsListBox.Items.Add($"Accuracy: {accuracy}");
```

The function, runUnigramTaggerButton_Click is triggered by pressing the “Run Unigram Tagger” button. A list, predictedTags, is found by using the Tag function on the unigramTagger class object created previously. After navigating to the token level of the testDataSet, the trueTag is determined and then the predictedTag is determined using the index value of the token of the true tag.

The predicted tag and the true tag are compared. If they are equal, true positives are incremented. If not, if the unigramTagger’s mostFreqTags is found to have the word but not the correct tag, false positives are incremented. Else, false negatives are incremented.

The precision, recall, F measure and accuracy are calculated and the results are printed.

## Statistic computed in Step 4 of part (a)

```
POS Tag Counts:
DET:117090
NOUN:241680
ADJ:73936
VERB:150590
ADP:126422
.:101152
ADV:45994
CONJ:32195
PRT:23349
PRON:35610
NUM:13886
X:1053

POS Tag Fractions:
DET:0.121594214487251
NOUN:0.250976938741813
ADJ:0.0767801677541157
VERB:0.156382891447905
ADP:0.131285197573723
.:0.105043111997732
ADV:0.0477632957650238
CONJ:0.0334334762611415
PRT:0.024247188607591
PRON:0.0369798443751902
NUM:0.0144201662171831
X:0.00109350677133039

Word-to-POS Tag Count:
Words with 1 different POS tags:42464
Words with 2 different POS tags:2780
Words with 4 different POS tags:29
Words with 3 different POS tags:186
Words with 5 different POS tags:3
Words with 6 different POS tags:1
```

From step 4 of part a, I found that the POS Tag with the highest count is NOUN, which makes sense given that nouns appear multiple times in the same sentence. The POS Tag with the lowest count is X, which also makes sense given that Other Tags are rarely found.

The POS tags' count in descending order is as follows: NOUN, VERB, ADP, DET, ., ADJ, ADV, PRON, CONJ, PRT, NUM, X

I also found that as the number of POS tags increases, there are fewer words which satisfy the criteria.

## Statistic computed in Step 6 of part (a)

```
True Positive: 158763
False Positive: 9363
False Negative: 6369

Precision: 0.944309624924164
Recall: 0.961430855315747
F-measure: 0.952793331292872
Accuracy: 0.909842688902261
```

From the results of step 6 of part (a), I found that rounded to 2 decimal places, Precision is 0.94, Recall is 0.96, F-measure is 0.95 and accuracy is 0.91

## Statistic for the final step of Part (b)

```
True Positives: 154211
False Positives: 21830
False Negatives: 0
Precision: 0.8759947966666856
Recall: 1.0
F1 Score: 0.9338989620047722
Accuracy: 0.8759947966666856
```

Figure 5: Results of final step of part (b)

From the results of final step of part (b), I found that rounded to 2 decimal places, Precision is 0.88, Recall is 1.00, F-measure is 0.93 and accuracy is 0.88

From the results, we can see that despite its complexity, the Perceptron tagger performs worse than the Unigram tagger, in terms of accuracy measured with F-measure