Intelligent Agents Assignment 1

Per-Ola Gradin

February 2025

Contents

1 Problem 1.1

In this problem we investigate parts-of-speech tagging by evaluating a unigram tagger implemented in C#, and a perceptron tagger implemented in python using the NLTK library.

1.1 a): Unigram tagger in C#

Below is a brief description of the code implemented within the given skeleton to complete the implementation of the unigram tagger.

POS Tag conversion

For the tag mapping, I chose to use a Dictionary¡string, string¿ with the Brown Tag as the key, and the Universal tag as the value. This is done using an instance variable

```
private Dictionary<string, string> tagMapping = new Dictionary<string, string>();
```

in the MainForm class. This is loaded by taking the BrownToUniversalTagMap.txt file and writing the Brown tag as the key, and the universal tag as the value in TagMapping. The conversion is done with the method ConvertPOSTags(tagMapping) within the POSDataSet class.

Splitting the data set

Splitting the data set in to a training set and a test set is done by first loading the split fraction from the GUI input, and then calling upon the static method POSDataSet.Split(completeDataSet, splitFraction) which simply returns the two new data sets. The split is done using the Take(splitIndex) and Skip(splitIndex) methods.

Generating statistics

Firstly, we generate statistics based on the fraction and absolute count numbers of the different universal tags found in the training data set. This is done via the instance method CountPOSTags() which returns a dictionary containing the tags, their count and their fraction of all tags. The results are shown in Table 1.

Universal Tag	**Count**	**(%) of total**
NOUN	241 680	25.10%
VERB	150 590	15.64%
ADP	126 422	13.13%
DET	117 090	12.16%
.	101 152	10.50%
ADJ	73 936	7.68%
ADV	45 994	4.78%
PRON	35 610	3.70%
CONJ	32 195	3.34%
PRT	23 349	2.42%
NUM	13 886	1.44%
X	1 053	0.11%
Total	962957	100%

Table 1: Frequency distribution of Universal POS tags in the training data set.

Secondly, we look at the distribution of number of tags per word. Most words only occur with one tag, but some words are associated with several tags, for instance "bear" which can be a noun or a verb. We generate this statistics by calling the method trainingDataSet.CountTagsToWordDistribution(), which works similarly to countPOSTags but returns a dictionary containing the number and fraction of words associated with 1, 2, 3... different tags. The results are shown in Table below.

Number of Tags	**Word Count**	**(%) of total**
1	42 464	93.40%
2	2 780	6.11%
3	186	0.41%
4	29	0.06%
5	3	0.01%
6	1	0.00%
Total	45 463	100%

Table 2: Distribution of words based on number of different assigned POS tags.

Unigram tagger

In order to generate the unigram tagger, I implemented a new class UnigramTagger, that is inherited from the POSTagger class. The class has an instance variable

```
private Dictionary<string, TokenData> unigramModel
```

that is generated when an instance of the class is created. It's constructor takes a data set, in this case the training data, and the unigramModel dictionary is written with each word spelling as the key, and a token consisting of the spelling and the most frequent tag as the value. This could have been more effective by having some data structure containing each word and their different tags with their frequencies, in order to not count

the same kind of statistics again as we did in the generateStatistics method. However, the calculations are done rapidly so for this task, it works well as it is. The tagger has an override of the Tag method of the POSTagger class, which takes a sentence as input and returns a list of strings containing all the tags predicted by the unigram tagger. If a word is unknown, i.e. it was not found in the training data, the corresponding tag would be "UNKNOWN", but it would still be evaluated as a faulty tag prediction.

Unigram tagger results

After generating the unigram tagger using the first 80% of sentences, it was tested using the test data set with the remaining 20% of sentences. The accuracy was 0.9098 or approximately 91%, with 158 763 correctly tagged words out of a total of 174 495 words.

1.2 b): Perceptron tagger in python

In this part, we used and evaluated a pre-trained perceptron tagger from the NLTK library in python, using the same data as for the unigram tagger. The perceptron tagger achieved an accuracy of 0.8771 or approximately 87.7% which is noticeably lower than the unigram tagger. One explanation for this could be that the unigram tagger is generated using training data from the same data set as the test data, while the perceptron tagger is not trained on the same data set, at least to our knowledge. It has been mentioned both in the course lectures and relevant literature, that the training data can significantly effect results, for instance if a model is trained on very old data and tested on a newer data set, since language is always changing.

2 Problem 1.2

In this problem, we implement a perceptron classifier using C#, in order to train and evaluate it on data sets containing airline reviews. The classification is either a positive review (1), or a negative review (0).

2.1 C# Implementation

This section contains a brief description of the code implemented in the assigned skeleton for the project.

Tokenization and vocabulary generation

We use the Tokenizer class to tokenize the texts, one TextClassificationDataItem at a time, and add them to the item as a list of Token objects. The difficulty with tokenizing the texts correctly was to separate correctly, especially in the instance of abbreviations where dots are not the stop of a sentence. In order to do this, I pre-defined a HashSet containing some common abbreviations:

```
private HashSet<string> abbreviations = new HashSet<string>
{
    "e.g.", "i.e.", "mr.", "mrs.", "dr.", "etc.", "st.", "rd.", "ltd.", "co.", "inc.", ↩
        "jan.", "feb.", "mar.", "apr.", "jun.", "jul.", "aug.", "sep.", "oct.", "nov.",↩
         "dec.", "a.m.", "p.m.", "u.s.", "u.k."
};
```

Then, when Tokenizer runs, we split the string using Regex and a pattern that decides where to split, which will mostly be at spaces and commas. We then loop over each split string, ignoring empty strings, and checking if the string contains a dot in order to handle different cases. Here we use the HashSet abbreviations, in order to not split them, and also check for decimal numbers in order to keep them as one string. The strings are then added to a list of strings, and finally turned in to a list of Tokens which are returned when Tokenizer finishes.

We tokenize all the data sets, and then generate the vocabulary using the training data. This is done by the AddTokens method in the Vocabulary class, which adds the spelling of the word and the token as a dictionary within the Vocabulary class.

Perceptron classifier

The perceptron classifier is implemented in the PerceptronClassifier class, which extends the TextClassifier base class. It is initialized with random weights for each token, with the weights stored in a

```
private Dictionary<string, double> weightDictionary;
```

and a double bias. The bias was left unchanged, since there were no instruction for it and it also seemed to not affect results significantly when testing. The classifier has an override method Classify, which calculates the sum of the weights associated with the input Tokens and the bias, and returns 1 if the sum is larger than 0, otherwise it returns 0. It also has methods for getting and setting weight and bias, and storing and loading the best performing (on the validation set) weights and bias. Further, it has two methods for getting and saving the 10 words with the highest and lowest weights, and saving some text classification examples respectively. This is used only for evaluating the classifier for the report.

Perceptron optimizer

In order to train the perceptron, we use the class PerceptronOptimizer, which is initialized with the perceptron classifier, the training data and the validation data. It is trained by running the Train method, which contains a while loop running until a boolean stopRequest is set true, and until then running the TrainOneEpoch method. This method trains the perceptron by shuffling the data set, then iterating the learning rule

$$\omega_j \leftarrow \omega_j + \eta(d_i - \Theta(y_i))x_{i,j} \tag{1}$$

as instructed. After updating all weights, we evaluate the perceptron using an instance of the PerceptronEvaluator class to assess its accuracy on the training and validation sets. If the validation accuracy is higher than the previous best accuracy, we store the new

weights and bias with the SetBest method of the PerceptronOptimizer class. After one epoch is completed, we trigger an event using an Action delegate

```
public event Action<int, double, double, double> EpochCompleted;
```

invoked at the end of TrainOneEpoch in order to update the GUI in a thread-safe manner. The event is handled in the MainForm class when initializing the optimizer by assigning a lambda function to the EpochCompleted event when initializing the optimizer:

```
perceptronOptimizer = new PerceptronOptimizer(classifier, trainingSet, validationSet);
perceptronOptimizer.EpochCompleted += (epoch, trainingAccuracy, validationAccuracy, ↩
    bestValidationAccuracy) =>
{
    ShowProgressSafe($"Epoch {epoch}: ", clearBefore: true);
    ShowProgressSafe($"Training accuracy = {trainingAccuracy:F4},\t Validation accuracy↩
        : {validationAccuracy:F4},\t Best validation accuracy: {bestValidationAccuracy:↩
        F4}");
};
```

Which uses the ShowProgressSafe method to update the GUI in a thead-safe manner. The optimizer is stopped by calling the method PerceptronOptimizer.Stop() from the stopOptimizerButton_Click method in MainForm. The optimizer also includes the method SaveTrainingData, which stores the training and validation accuracy from each epoch in a CSV file for further analysis.

Perceptron evaluator

The PerceptronEvaluator is a simple class initialized with a classifier, and evaluating the accuracy of said classifier using the Evaluate method, taking a TextClassificationDataSet as input. It simply counts the amount of correctly labeled TextClassificationDataItems, divided by the total number.

Noteable MainForm implementations

To prevent the GUI from freezing during training, we execute the optimizer in a separate thread:

```
optimizerThread = new Thread(() =>
{
    perceptronOptimizer.Train();
});
```

The ShowProgressSafe method ensures that GUI updates occur in a thread-safe manner. If the method is called from a background thread, it invokes the GUI thread to update the progressListBox. If it is already running on the GUI thread, it updates the list directly. One issue encountered was that the optimizerThread.Join() call could cause a deadlock: the main thread waited for the optimizerThread to finish, while the optimizerThread was blocked waiting for the GUI thread to process updates in ShowProgressSafe. This issue was resolved by offloading the thread-joining process to a background task using Task.Run.

2.2 Results

After training, the perceptorn classifier reached the results shown in table 3. This was obtained after loading the best weights and bias found during training, as evaluated on the validation data.

Training accuracy	**Validation accuracy**	**Test accuracy**
0.9970	0.9600	0.9750

Table 3: Final results of the perceptron classifier.

In figure 1, we can see the accuracy on the training and validation data during the training process.

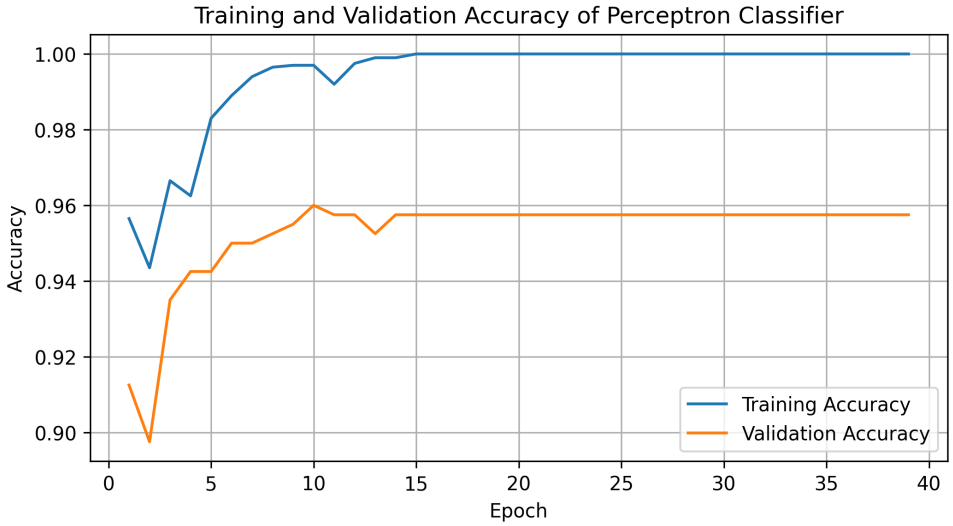

Figure 1: Training and validation accuracy over the first 40 epochs of training.

We can see that the training accuracy reaches 100% at 15 epochs, after which it no longer updates the weights. We can also note that the validation accuracy peaks before this point, and the stored best weights and bias outperform the final weights found during training. This highlights the importance of storing the weights with best performance on the validation set.

For further analysis, we can see the top 10 most positive and negative words based on their weight values in table 4.

Most Positive Words	Weights
Excellent	6.5803
Good	6.5511
Great	6.1372
Thank	5.3448
Comfortable	4.9931
Best	4.2849
Well	4.2762
Friendly	3.9015
Did	3.8338
Airlines	3.7795

Most Negative Words	Weights
No	-6.8981
Never	-6.8318
Rude	–6.7514
Told	-5.9137
Worst	-5.0360
Not	-4.7799
Hours	-4.6952
They	-4.3257
Customer	-4.2576
Poor	-4.1851

Table 4: Top 10 most positive and most negative words based on weight values.

On the positive side, most of the words are quite intuitive, aside from perhaps "did" or "airlines". Words like "excellent" and "great" can feel rather obvious as words with a positive connotation. However on the negative side, we see more words that are not as obviously related to negative views, at least intuitively, such as "hours", "customer" or "told". Even the top negative word "no" is a very common word and can be used in a variety of ways in positive remarks. However, in the context of airline reviews, many of them become more understandable. For example, it is reasonable to assume that "hours" are mostly brought up when someone has been waiting for hours due to delays, and most positive reviews wouldn't mention it.

Correct classification examples

Here we look at two correctly classified examples of airline reviews.

> Text: "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!", Predicted class: 0, Actual class:0
>
> Text: "i took a flight from dublin to toronto with air transat over the summer. this is my fourth year in a row with the company and they never fail to disappoint. staff are friendly and the aircraft spotless. definitely value for money.", Predicted class: 1, Actual class:1

We can see that the first example is a really long review, with a lot of negatively loaded words and many instances of the words in the top 10 negative words from table 4. For instance, "hours" as we mentioned before, is mentioned more than once. The second example however is a short review, with a few words that could many times be associated with negative sentiment, such as "fail" and "dissappoint". However there are words such as "friendly" and "spotless" that would be positive in most cases. Therefore, it is quite interesting that the classifier was able to classify this review correctly.

Incorrect classification examples

Here, we look instead at examples of when the classifier failed.

> Text: "sunday 21st june, shanghai to mexico city premier class. old aircraft with no inflight entertainment this situation bother me so much. there was also a complete lack of interest shown by the cabin crew in their jobs. very disappointing. the flying experience was not be very good on premier class after i bought an upgrade from economy class. ", Predicted class: 1, Actual class:0
>
> Text: "super convenient 3 pm departure from delhi. only direct flight from delhi to madrid. excellent legroom in economy class - more than any other long-haul airline i have been on. adequate film choices in english. decent meal, wholesome snack - more substantive than ana which i flew to tokyo from delhi last month. absolutely nothing wrong in traveling this airline. asked for a gin and tonic, got two without asking. just don't understand why people criticize this airline so much. it's my preferred choice in economy when possible.", Predicted class: 0, Actual class:1

The first example here contains some obvious negative sentiment such as "very dissappointing", but there are some misleading words for the classifier such as "very good", which can lead to a faulty classification. There aren't a lot of occurencies of top 10 most negative or positive words in this example, which could also affect how well the perceptron is able to classify it. In the second example we see a similar situation, but with a longer text. Here it seems many of the positively connotated words are not in the top 10, and could potentially be uncommon in the training set. This could be an explanation of the miss-classification, and for similar examples. If a review uses uncommon words with a strong positive or negative sentiment, it would most likely prove more difficult for the perceptron classifier.

3 Problem 3

In this problem, we implement a naïve bayesian classifier using C# in order to classify restaurant reviews, similar to the perceptron tagger in problem 2.

3.1 C# Implementation

This section contains a brief description of the code implemented in the provided skeleton.

Tokenization and vocabulary generation

The tokenizer implemented was reused from problem 2, so it will not be described further here. There was however a minor change in the vocabulary generation, with the instance variable `vocabulary` being

```
private Dictionary<string, TokenData> vocabulary;
```

where we now use `TokenData` as the entry Value instead of token. This allowed for using `Count`, `Class0Count` and `Class1Count` when building the vocabulary. This data is later on used in the `NaiveBayesianClassifier` class as a "Bag of Words".

Naive Bayesian Classifier

The class `NaiveBayesianClassifier` was created in order to create the classifier, and also classify text items and evaluate performance. It is initialized with dictionarys containing the posterior likelihoods for both classes, with the dictionary key as the different words spelling, and prior likelihood for each class as well as the vocabulary as mentioned previously. It uses the vocabulary, generated by the training data, to upon initialization compute the posterior and prior using the `ComputeProbabilities` method.

The prior is calculated as

$$\hat{P}(c_j) = \frac{count(class = c_j)}{m} \tag{2}$$

where m is the total amount of reviews. The posterior is calculated as

$$\hat{P}(w_i|c_j) = \frac{count(w_i, c_j) + 1}{\nu + \sum_{k=1}^{\nu} count(w_k, c_j)} \tag{3}$$

where ν is the amount of distinct words in the vocabulary, which becomes easier with the vocabulary already containing the counts of a words occurrence in the different classes. We use Laplace smoothing in eq. (3) in order to not get zero-entries.

The probabilities are stored, and then used in the `Classify` method in order to classify a review. It takes a `TokenList` as input, and gives an output integer denoting the predicted class of the review. This is done using the equation

$$\hat{c} = argmax_{j \in 1,2,..,k} \left(log(\hat{P}(c_j)) + \sum_{i=1}^{\nu} log(\hat{P}(w_i|c_j) \right) \tag{4}$$

looping over the words present in the review.

We further have the method `Evaluate`, taking input in the form of a data set, and calculating accuracy, recall, precision and F1 measures of the classifier. We also have a methods dedicated to accessing the priors $\hat{P}(c_j)$, and calculating the probabilities $\hat{P}(c_j|w_i)$. The latter is calculated using Baye's rule as in the equation

$$\hat{P}(c_j|w_i) = \frac{\hat{P}(w_i|c_j)\hat{P}(c_j)}{\hat{P}(w_i)} \tag{5}$$

where we already have the prior $\hat{P}(c_j)$. We calculate the probabilities for a word occuring as

$$\hat{P}(w_i) = \hat{P}(w_i|c_0)\hat{P}(c_0) + \hat{P}(w_i|c_1)\hat{P}(c_1) \tag{6}$$

which is the same as the count of a word divided by the total count of words. For the posterior, we use precomputed values stored in dictionaries. These are computed using equation (3), but without Laplace smoothing. These non-smoothed posteriors are calculated alongside the smoothed-out ones in the `ComputeProbabilities` method. The calculations of $\hat{P}(c_j|w_i)$ are done in the methods `GetPosteriorPositive` and `GetPosteriorNegative` methods, which have as input the word w_i.

Additional code

The `MainForm` was kept quite simple. I implemented buttons for generating the vocabulary, generating the model and evaluating the model respectively. Additionally, a restart button was added, in order to simplify the process of switching data sets and evaluating on the airline reviews for instance. In the `evaluateButton_Click` method, the different measures and statistics are printed out in the `progressListBox`. Since the program ran quickly and had no need for parallel threads, it was not implemented.

3.2 Results

The naïve bayesian classifier was evaluated on both the restaurant review data provided, and also the airline reviews used for the perceptron classifier in problem 2. The prior probabilities computed from the training set are:

$$\hat{P}(c_0) = 0.49, \quad \hat{P}(c_1) = 0.51 \tag{7}$$

where c_0 represents negative reviews and c_1 represents positive reviews. To further analyze how the classifier interprets specific words, we examine the posterior probabilities $\hat{P}(c_0|w)$ and $\hat{P}(c_1|w)$ for four selected words: "friendly", "perfectly", "poor", and "horrible", seen in table 5.

Word	$\hat{P}(c_0\|w)$ (Negative)	$\hat{P}(c_1\|w)$ (Positive)
friendly	0.1081	0.8919
perfectly	0.0000	1.0000
poor	1.0000	0.0000
horrible	1.0000	0.0000

Table 5: Posterior probabilities of class labels given the presence of selected words.

The results confirm that "friendly" and "perfectly" are strongly associated with positive reviews, while "poor" and "horrible" are exclusively found in negative reviews. Notably, three out of four words only occur in one of the two classes. This demonstrates how the classifier relies on word presence and absence to determine sentiment.

The classification performance was evaluated using accuracy, precision, recall, and F1-score for both datasets, as in the compendium 4.5.1. In table 6, we can see the results obtained on the different datasets.

Dataset	**Accuracy**	**Precision**	**Recall**	**F1-score**
Restaurant Reviews (Training)	0.9344	0.9464	0.9237	0.9350
Restaurant Reviews (Test)	0.7800	0.7209	0.7561	0.7381
Airline Reviews (Training)	0.9670	0.9736	0.9600	0.9668
Airline Reviews (Test)	0.9350	0.8991	0.9800	0.9378

Table 6: Performance metrics for the restaurant review dataset and the airline review dataset.

The Naïve Bayes classifier achieved high accuracy on the training sets, with 93.44% for restaurant reviews and 96.70% for airline reviews. However, performance on the test sets was lower, particularly for restaurant reviews, where accuracy dropped to 78.00%. One reason for the drop in test accuracy is likely due to the assumption of word independence in Naïve Bayes. Many words in natural language have dependencies, and breaking these dependencies can lead to errors in classification. Additionally, some words present in the test set may not have been seen in the training data, causing them to be ignored in the classification process.

The performance on the airline dataset is significantly better, with 93.50% accuracy on the test set. This suggests that airline reviews might contain more distinguishable patterns compared to restaurant reviews.

We can also note some difference in size and character of the datasets. The restaurant review dataset contained 10 881 tokens in the training data and 1 643 tokens in the test data, with a vocabulary of 1 958 unique tokens. Since the test data contained 100 reviews, it has an average of around 16 tokens per review, meaning they are rather short. For the airline reviews, we have 284 881 tokens in the training data and 30 431 tokens in the test data, with a vocabulary of 12 411 unique tokens. We can thus see a substantial different in the length of the reviews, which could have a large impact on the difference in

performance. Longer reviews provide more context per instance, which could be making it easier to classify sentiment accurately, especially since it is a statistical model.

4 Problem 4

In this problem, we implement an auto-complete system using n-grams. It automatically suggests the next word based on previous words, if there are any.

4.1 Implementation

This section provides a brief description of the C# implementation for the auto-complete system.

Dataset handling and Tokenization

The first step of the implementation was data gathering, which consisted of ten different books from the Gutenberg Project (available at `https://www.gutenberg.org/`), and movie dialogs from the Cornell Movie-Dialogs corpus. The data was preprocessed by removing some book-information by hand, and some data cleansing in python. The tokenization process was very similar to that which was implemented in problems 2 and 3, with one difference being that we saved the tokens in a `List<string>` datatype. There were additional minor changes, but nothing substantially. The data consisted of approximately 7.8M tokens. Since the tokenization process took more time for this problem, with the larger dataset, it was ran in a separate thread from the GUI.

N-gram generation

After tokenization, the program generates n-grams (unigrams, bigrams, and trigrams) using an instance of the `NGramManager` class. This class stores a `Dictionary<string, List<NGram>>` for each n-gram, which are created using the `GenerateNGrams` method. This process involves looping over every token and storing itself, and $n-1$ following tokens, in `Dictionary<string, int>` datatypes in order to keep track of the count for each n-gram. They are then converted to `Dictionary<string, List<NGram>>` where for each `NGram` the frequency per million tokens was stored. This builds on a maximum likelihood estimate

$$\hat{P}(t_1, t_2, t_3) = \frac{count(t_1, t_2, t_3)}{m} \tag{8}$$

where t denotes a token, and m denotes the total number of n-grams. In order to avoid freezing the GUI, the generation of n-grams was handled by a separate thread, showing progress with the method `ShowProgressSafe` in a thread-safe manner.

In table 7, we can see the total count of the different n-grams.

N-gram Type	Count
Unigrams (total unique tokens)	90173
Total bigrams	1268786
Total trigrams	3999499

Table 7: Total number of n-grams in the dataset after preprocessing and tokenization.

AutoCompleter class

The `AutoCompleter` class is responsible for generating word suggestions based on the previous words, using an instance of the `NGramManager`. The core function, `GetSuggestions` takes a list of tokens (words typed so far) and attempts to predict the next word based the most frequently occuring n-grams. It prioritizes trigrams:

```
if (tokens.Count >= 2)
{

    string lastTwoWords = $"{tokens[tokens.Count - 2]} {tokens[tokens.Count - 1]}";
    if (nGramManager.GetTrigramDictionary().TryGetValue(lastTwoWords, out var ↩
        trigramList))
    {
        return trigramList
            .OrderByDescending(ngram => ngram.FrequencyPerMillionInstances)
            .Select(ngram => ngram.TokenList.Last())
            .Distinct()
            .ToList();
    }
}
else if (tokens.Count == 1)
{

    string searchKey = $". {tokens[0]}";
    if (nGramManager.GetTrigramDictionary().TryGetValue(searchKey, out var trigramList)↩
        )
    {
        return trigramList
            .OrderByDescending(ngram => ngram.FrequencyPerMillionInstances)
            .Select(ngram => ngram.TokenList.Last())
            .Distinct()
            .ToList();
    }
}
```

It does so by both checking if there are two words present, and returning the most frequent trigrams, or if only one word is present, it checks for the most likely trigram beginning with a ".". If there are none, it will fall back on bigrams.

Mainform logic

The `MainForm` class handles the GUI and user interaction. After loading a dataset, tokenizing the input and generating the n-grams, the user can click on the "Start Autocompletion" button. When there is a change in the input text box, the `sentenceTextBox_TextChanged` method uses the `AutoCompleter.GetSuggestions` to get suggested words, and puts them in the suggestion list box. If there are no suggestions, the suggestion list box stays invisible. The user can choose a suggested word either by clicking on an option, or pressing the TAB key on the keyboard to get the top suggestion instantly. In order to get smooth punctuation in the input text box, we use a check

```
string punctuation = ".,!?;:";
string selectedWord = suggestionListBox.SelectedItem.ToString();
if (punctuation.Contains(selectedWord))
{
    sentenceTextBox.Text = sentenceTextBox.Text.TrimEnd() + selectedWord;
}
else
{
    sentenceTextBox.Text = sentenceTextBox.Text.TrimEnd() + " " + selectedWord;
}
```

avoiding unnecessary blank spaces between words and punctuation marks.

In order to not lose focus of the input text box, we use the method `sentenceTextBox.Focus();` so the user won't have to re-click on the text box to continue typing.

4.2 Discussion

Since this auto-complete system is rather hard to evaluate, the discussion will be brief. However, one can see a few interesting patterns and features by using it and testing around. One thing I found was that it can often get stuck in a loop when just selecting the top suggested word, and it will produce nonsensical sentences. However the suggestions for one word can often seem reasonable. Since we at most look at trigrams, it should be expected that this auto-complete system has a hard time producing a longer sentence which makes sense, since it doesn't really have data for it. If we look at table 8, we can see the top 5 most common trigrams.

Trigram	**Frequency per million**
`".", "well", ","`	530.07
`".", "i", "don't"`	475.77
`"i", "don't", "know"`	470.16
`",", "and", "the"`	357.37
`".", "oh", ","`	347.68

Table 8: Top 5 most frequent trigrams in the dataset, ranked by frequency per million instances.

This matches well with my user experience, where I very often saw it get stuck in a loop of producing "I don't know" when taking the top suggestion several times.

One should also say that with statistical models like this one, the dataset can have a large impact on results. In this case, since it consisted of a large amount of books, and many of them older, it can have an influence on formality, outdated words and thus word suggestions. If it were to be used for, let's say text messaging or a search engine's search bar, it would probably be better with a different set of data.