TME286 - Homework Problem 1

Intelligent agents

Felix Waldschock
000420-T398

Chalmers University of Technology
February 2024

Contents

1 Problem 1.1: Part-of-speech (POS) tagging

In this problem, words respectively tokens, shall by tagged a universal tag such as "Noun" or "Verb". For this a Unigram tagger is implemented and trained with a portion of the given data set, afterwards it is evaluated with the remaining data.

1.1 Part a): C#-Implementation

Following the implementation in of the software in C is covered.

1.1.1 C# Implementation

Converter
As the dataset comes with Brown tags and the Unigram tagger shall be trained with universal tags, a conversion of these tags is necessary. For this purpose a new class `POSTagCoverter` is created. When initialized a dictionary is created

```
Dictionary<string,string>
```

This dictionary stores the Brown and Universal tag pairs (e.g. NN -¿ NOUN). The class also contains a method `getUniversalTag(string Browntag)` which returns the Universal tag that corresponds to the Brown tag. With this method the data set is converted from Brown to Universal.

Splitting of dataset
The given data needs to be split into a training and a test set, to have certain data for the creation of the Unigram tagger and some data to classify its accuracy. To do so, in the `POSDataSet` class a new method is implemented which takes the dataset itself and a `double splitFraction` .

```
1    numberOfSentences = POSDataSet.SentenceList.Count
2    splitIndex = (int) numberOfSentences * splitFraction
```

Now the `SentenceList` is split into two lists at the index `splitIndex`. The two generated lists are then used to initialize new `POSDataSet` objects.

WordData Class
The `WordData` class is a key class of the entire software. For each spelling an instance of this class generated, and in stores the different tags and also their number of occurrences in it. The class comes with to methods, one to add a tag to a spelling, the other to get most frequent tag the spelling has in the dataset. The class comes in very handy for the creation of the Unigram tagger, as the most frequent tag for a spelling is stored in it, but also for the generation of the basic statistics as the dictionary also contains the information of how many tags are linked to a spelling.

```
1    public Dictionary<string, int> TagsCount { get; } = new Dictionary
     <string, int>();
2    public void AddTag(string tag)
3    public string GetMostFrequentTag()
```

Statistic generation
To get a basic understanding of the data set a basic analysis of it is conducted. Firstly the distribution of the Universal tags is observed. As in Table 1 visible, nouns are represented the most, with about 24% followed by verbs with 16%.

Universal tag	**Count**	**Fraction**
NOUN	219967	0.2420
VERB	146379	0.1610
ADP	115876	0.1275
DET	109554	0.1205
.	98376	0.1082
ADJ	67153	0.0739
ADV	44953	0.0494
PRON	39540	0.0435
CONJ	30416	0.0335
PRT	23914	0.0263
NUM	12000	0.0132
X	970	0.0011
$\sum$	909098	1.0000

Table 1: Distribution of Universal tags in the training set, where the training set is containing the first 80% of the initial dataset

Secondly the commonness of a spelling having multiple tags assigned is analyzed. The results are shown in Table 2. We find that 94% of the words have either one or two tags. The four words that have more than four tags are shown in Table 3.

n-Tags	**Count**	**Fraction**
1	41576	0.9327
2	2777	0.0623
3	188	0.0042
4	32	0.0007
5	3	0.0001
6	1	0.0000
7	0	0.0000
8	0	0.0000
9	0	0.0000
10	0	0.0000
11	0	0.0000
12	0	0
$\sum$	44577	1.0000

Table 2: Number of tags assigned to a spelling

Spelling	Number of tags	Tags
well	5	ADV, ADJ, NOUN, PRT, VERB
damn	5	VERB, ADV, ADJ, PRT, NOUN
round	5	NOUN, VERB, ADJ, ADV, ADP
down	6	PRT, ADP, NOUN, ADV, VERB, ADJ

Table 3: The four words that were found having more than four assigned tags

POS Tagger class

The class `UnigramTagger` is inherited from the `POSTagger` class. The new class mainly exist of its dictionary which is generated with the initialization of the tagger. This dictionary consists of the string key `spelling` and contains `TokenData` items as its value. These `TokenData` items contain also the spelling and also the most frequent POSTag, which it gets with the help of the WordData objects, described above. The class overrides the method `Tag`

```
1 public override List<string> Tag(Sentence sentence)
```

of the `POSTagger` class. The method takes a sentence and loops over every token in this sentence. It tries to find a corresponding spelling in the `unigramTaggerWordDataDictionary` and returns a list of tags, which are of type string. Spellings which are not stored in the Tagger get the tag "UNKNOWNSPELLING" associated and are skipped in the evaluation.

1.1.2 Results of the Unigram tagger

To determine the quality of the created tagger, the *Test set* is used. The *Test set* contains the last 20% of sentences the original *Brown Corpus*. All tokens of the test set get combined in a new `List<TokenData>` and handled as a sentence. The tagger predicts for each token a Universal Tag, this prediction is then compared to the ground truth. The accuracy is then calculated as the number of correct matches and the total number of tokens in the sentence. The Table 4 below, shows the accuracy's for different split ratios (different ratios between the sizes of the training and test set).

Split	**Accuracy**
0.5	0.9518
0.6	0.9523
0.7	0.9523
0.8	0.9528
0.9	0.9531

Table 4: Different achieved accuracy's for varying split rations between training and test set.

The achieved accuracy of the tagger is around **95%**, and therefore classifies the words very well. Interestingly to see that the split ratio does not have a significant impact on the test. This can maybe be explained with the fact that the number of total tokens, 1'137'452, is compared to the number of Universal tags, 12, relatively high.

1.2 Part b): Python implementation

In this exercise the same text sample is analyzed, but with a **pretrained perceptron** tagger instead of the Unigram tagger. For this the *NLTK* library and *Google Colab* are used.
The perceptron tagger managed to reach an accuracy of **82.42%**, which is compared to the results of 1.1.2 lower. This can be explained with the fact that the perceptron is not trained with the particular data set.

2 Problem 1.2: Perceptron text classifier

In this problem a perceptron is trained to classify if a airline review is positive or negative.
To do so a binary perceptron text classifier is implemented in a C# project. The project contains three main difficulties.

1. Tokenizing the data
2. Indexing the input data and creating a vocabulary
3. Perceptron implementation

The implementation of these points shall be discussed in the following subsections.

2.1 Implementation

To do so a binary perceptron text classifier is implemented in a C# project. The project contains three main difficulties.

1. Tokenizing the data
2. Indexing the input data and creating a vocabulary
3. Perceptron implementation

The implementation of these points shall be discussed in the following subsections.

2.1.1 Tokenizing the data

The challenge in tokenizing the reviews, which come often as a set of senteces, is that using a *space* and *dot* character leads to incorrect tokens. As there are tokens such as abbreviations with with a dot (e.g.) or also numbers with decimals like (2.134).
To help out with this issue, the Regex capabilities of C# are used. In a first step the character set '?' '!' ',' and *space* is used as a delimiter in between tokens.
In a second step it is checked if the token contains a '.' character. The pseudo code for the handling of the tokens containing a '.' character:

```
if (token contains '.')
{
    if (token exists in abbreviation list)
    {
        token = token
    }
    else if (token is decimal number)
    {
        token = token
    }
    else
    {
        split token by '.' character
    }
}
```

Furthermore the tokenizer converts currency characters as {€ $ £} to their spelling {euro, dollar, pound}. With this adjustment a slight improvement of about 0.5% in accuracy could be achieved.

2.1.2 Indexing the data & vocabulary

After the training dataset has been read in the vocabulary is generated out of it. Therefore all, distinct, tokens (of the training data set) are placed in one list of, type token, and then sorted according to their spelling in alphabetical order. The position a token now takes in this sorted list, is also its index from now on. In the implementation of this method the `Distinct()` came very handy. Furthermore the `Vocabulary` class got the methods to find an index with a given spelling, and vice versa.
The indexing of the data takes a fast amount of computation time, in this particular case, around 16 minutes are needed to index all 2600 reviews.

2.1.3 Perceptron

The implementation of the perceptron is split into three classes. The classifier, the optimizer and the evaluator.
Perceptron classifier: The perceptron classifier contains the parameters of the such as:

- weights
- number of epochs
- best found weights
- best found accuracy

When the class is initialized, a list of double values with the size of the number of features (unique words) is created. The weights are set randomly to start. In the class two main methods are implemented `Classify` and `optimizePerceptron`. The method

```
1    public override int Classify(List<int> tokenIndexList)
```

takes a list of indexes, which represents a review, collects the corresponding weights and sums them up. This sum is then used as an argument to a sign-activation function and determinded if a review is classified as a positive (class = 0) or positive (class = 1) one. In the `optimizePerceptron` method, calls the `PerceptronOptimizer` class which is explained following.

Perceptron optimizer: In the perceptron optimizer class, the training of the perceptron is handled. It consists of the method `trainClassifier` and the variable `learningRate` which determines the "intenseness" of a weight update. The method

```
1    public void trainClassifier(PerceptronClassifier
     perceptronClassifier
2    , TextClassificationDataSet trainingDataSet)
```

does the following main mechanics:

- Shuffle the order of reviews in the training dataset
- Loop over each review and predict its class (with the `Classify` method of the `PerceptronClassifier`
- If ground truth != prediction $\rightarrow$ adjust the weights with:

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

where w_j is the weight, η is the learning rate, d_i the ground truth, Θ the sign activation function, y_i the review and $x_{i,j}$ the number of occurences of a feature (word) in a review.

Perceptron evaluator: In the perceptron evaluator class, as the name suggest, a data set can be evaluated with the perceptron. Therefore the method

```
1 public double evaluatePerceptron(PerceptronClassifier
     perceptronClassifier, TextClassificationDataSet dataSet)
```

is implemented. It takes as an argument the perceptron classifier and a dataset. It loops over all review in the data set and classifies them. Two counter variables keep track of the number of successes and the number of predictions. After the classification of the entire data set is completed the accuracy is computed as:

$$accuracy = \frac{\text{correctClassifications}}{\text{totalClassifications}}$$

2.1.4 Results

After training the perceptron with a learning rate $\eta = 0.1$ the following results are found. As in Table 5 listed, the accuracies of the perceptron are for both, the validation and the testing set, with over 95% fairly high and show that the perceptron is well performing on the classification task. The development of the accuracy, for the training and validation set, is shown in Figure 1. One finds that after already 15 epochs, the perceptron goes into a steady state.

Training	**Validation**	**Test**
1.000	0.965	0.960

Table 5: Accuracies of the perceptron on the 3 data sets, with the best performing weights

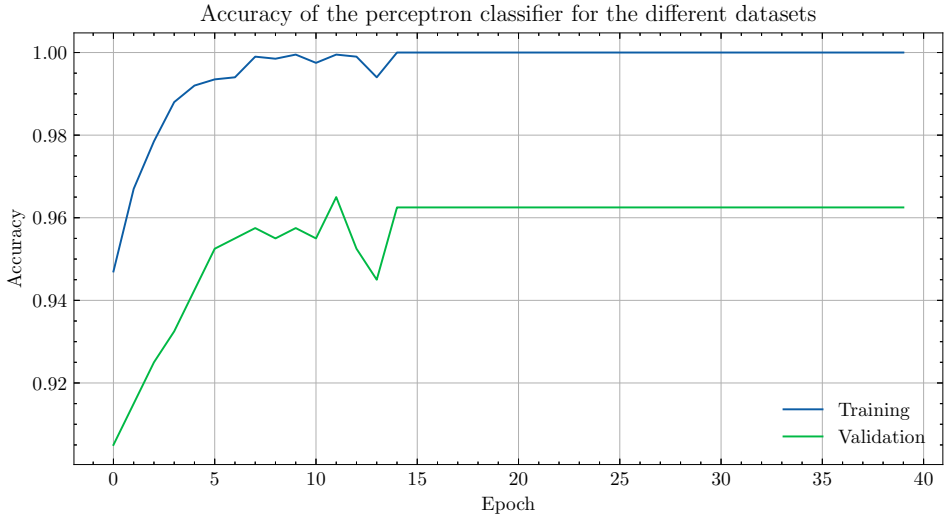

Figure 1: Accuracy of the classifier for the training and validation set

In Table 6 the 10 most positive and 10 most negative words are listed. For 9 of the 10 words a positive meaning seems obvious (e.g. good, excellent ...), differently for the word "crew", this finding seems interesting. For the 10 most negative words, a bit more "variety" is found. The words "or", "hours", "because" would, on a first glance, probably not be associated with a negative meaning.

Table 6: Extreme words, determined by their weights

(a) Most positive words

Word	**Weight**
good	7.8254
comfortable	7.5820
great	7.1578
best	6.0660
excellent	5.9794
thank	4.8798
very	4.5699
nice	4.8798
crew	4.5699
friendly	4.5316

(b) Most negative words

Word	**Weight**
never	-7.3594
no	-6.9300
worst	-6.1805
because	-5.7775
not	-5.0851
told	-5.0397
rude	-4.8362
hours	-4.5254
poor	-4.3102
or	-5.2602

Examples of correct classified reviews

```
Before boarding, the terrible customer service people at the gate made
    people check their bags even though there was an excessive amount
   of space in overhead compartments. With their terrible rate of
   baggage loss, this has ruined my flight and others.

London to Delhi. An excellent service and experience. This was my
   first time travelling with AI and I was amazed with service from
   ground staff to onboard. I'm looking forward flying with AI again
   in future.
```

Examples of falsely classified reviews

```
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.

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.
```

In the first example of falsely classified reviews, I could imagine that the keyword *not* in the last sentence of the review, has not a big enough emphasize on the classification. If one checks the sentence without *not*, the meaning becomes fairly positive. Also in the second review one finds, that the second last sentence contains a *don't* which has a big impact on the sentence, and may lead with the perceptron approach to a false classification.

3 Problem 1.3: Bayesian text classifier

In this problem a Bayesian text classifier is implemented to classify restaurant and airline reviews. The results are compared to the perceptron classifier from section 2.

3.1 Implementation

The main architecture of the software is very similar to the perceptron classifier 2. The main difference is, of course, the classifier class. As the Bayesian classifier does not need an iterative training. Following the `BaysianClassifier` class is briefly described. The class is initialized with the `Bag of words`, a list of `TokenData`, that stores each individual spelling of the training data and also the frequency of the spelling appearing in a positive or negative review. With the "Bag of Words" the Prior and Posterior term are computed. Where the Prior is computed as:

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

To simpflify this computation, the class `TextClassificationDataSet` were extended by the two variables:

- numberOfDocumentsOfClass0
- numberOfDocumentsOfClass1

They store the number of reviews that are associated with class 0 or 1.
For the Posterior the likely hood of a word being in a class 0 or class 1 review needs to be computed. Therefore the "Bag of Words" is again needed. All words, that are check but not contained by the "Bag of Words", neglected and return a word likelyhood $\hat{P}(w_i|c_j) = 1$. This since, when the *log* is later used, for better handling of the small values, we get a contribution of 0.
To determine the word likelyhood, the equation 4.26 of the Compendium is used. This formula also contains the **Laplace smoothing**.

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

3.2 Results

As well as already with the perceptron in the section before, also fairly well results were obtained by the Bayesian classifier. In the following Table 7 the classifier is evaluated according to the evaluation measures in the Compendium (4.5.1). Precision measures the accuracy of the positive predictions made by the model. We find that for the airline reviews, it is about 6% lower than for the restaurant reviews. The recall measures the ability of the model to capture all the relevant instances of the positive class, here we find contrariwise a higher value for the airline reviews. The accuracy is a measure of the overall correctness of the model, which is for both datasets with over 92% fairly good. But we find that the classifier is outperformed by the perceptron. The F1 value provides a balance between precision and recall, making it a useful metric when there is an uneven class distribution or when both false positives and false negatives need to be minimized. In this particular case, its similar for both datasets.

	Restaurant review	**Airline review**
Precision	0.9743	0.8890
Recall	0.9268	0.9700
Accuracy	0.9600	0.9250
F1 value	0.9500	0.9280

Table 7: Evaluation measures

The Table 8 below, shows the likelyness of the four words "poor", "horrible", "friendly", "perfect" to be in either a positive or negative sentence. Both for the restaurant and also the airline reviews. Furthermore the ration between a words likeliness of being in a positive or negative review is computed. One finds for the word "poor", "horrible" in both cases a strong likelihood to be classified as negative, and vice versa for "friendly", "perfect". Interesting to see is that the ratios in the restaurant class are of similar amplitude, whereas in the airline reviews one finds ratios for "horrible", "friendly" of about 4-5 times higher. This ratio shows that the distribution of these two words is strongly different in the airline than in the restaurant reviews.

	Restaurant negative	**Restaurant positive**	**Ratio**	**Airline negative**	**Airline positive**	**Ratio**
poor	0.000815	0.000124	6.6	0.000429	0.000079	5.4
horrible	0.000815	0.000124	6.6	0.000514	0.000018	28.6
friendly	0.000466	0.002977	6.4	0.000117	0.00255	21.8
perfect	0.000116	0.000496	4.3	0.000032	0.000123	3.8

Table 8: Likelyhood of the words "poor", "horrible", "friendly", "perfect" to be in class 0 / 1 for the two datasets (restaurant & airline reviews)

Additionally the probabilities $\hat{P}(c_i|w)$ are computed. These probabilities are not computed on the run time, but can be, thanks to the Bayesian rule, determined. Therefore we use the Ansatz:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{4}$$

This leads to

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

All probabilities on the right hand side are known and given as:

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

with m = number of reviews. It is simply the prior. To words probability $P(w_i)$ computes as:

$$\hat{P}(w_i) = \frac{\text{count}(w_i)}{\text{total number of words}}$$

and the likelyhood of a word appearing given a class $\hat{P}(w_i|c_j)$:

$$\hat{P}(w_i|c_j) = \frac{\text{count}(w_i, c_j}{\sum_{k=1}^{v} \text{count}(w_k, c_j)}$$

This then leads to the values in the Table 9. For 3 of the four cases, the entire probability lays towards one class, as they only show up in this classes.

	Class 0	**Class 1**
friendly	0.102	0.897
poor	1	0
horible	1	0
perfectly	0	1

Table 9: Likelyhood of the two classes given a word

The size of the "Bag of Words" was for the restaurant reviews 2160 and for the airlines 12685.

4 Problem 1.4: Autocompletion with n-Grams

in this problem a software shall be developed, which automatically suggest the next word to a user. This next word is selected by checking which tri- or bigram is the most likely one, according to the current input.

4.1 Implementation

4.1.1 Tokenization

The tokenization is similar as in 2 and 3 and not explained in exact detail. What needs to be pointed out is that, as the dataset is fairly large compared to before, handling of special cases, characters, numbers and abbreviations is skipped. This leads to suitable tokenization times for the task.

4.1.2 Generation of the n-Grams

For the three n-Grams with $n \in [1, 2, 3]$, three different dictionaries are used. The contain of the key `identifier`, which is a string, and the value `nGram`, which is an instance of the class NGram. These 3 dictionaries are stored inside the `NGramsDataSet` class, where they are also generated. For their generation one loops over the list of tokens, which is generated by the tokenizer. To keep track of the number of occurences of a n-gram, a helper dictionary is used (example for computation of unigrams): For each token it is checked, if the dictionary already contains it, if so the counter is incremented, otherwise a new key and value is added to the dictionary. Following the code snipped for generating the Unigram dictionary.

```
unigrams = new Dictionary<string, NGram>();
Dictionary<string, int> unigramsCounter = new Dictionary<string, int
    >();
foreach (List<string> TokenList in listOfUnigrams)
{
    string identifier = string.Join(" ", TokenList);
    if (unigrams.ContainsKey(identifier))
    {
        unigramsCounter[identifier]++;
    }
    else
    {
        // create a new NGram
        NGram nGram = new NGram(identifier);
        nGram.TokenList = TokenList;
        unigrams.Add(identifier, nGram);
        unigramsCounter.Add(identifier, 1);
    }
}
// add the frequency per million to the NGrams
int totalNumberOfOccurencesOfAllUnigrams = unigramsCounter.Values.Sum
    ();
foreach (KeyValuePair<string, NGram> entry in unigrams)
{
    entry.Value.FrequencyPerMillionInstances = 1000000 * (double)
    unigramsCounter[entry.Key] / totalNumberOfOccurencesOfAllUnigrams;
}
```

When the execution of looping over all tokens is finished, the `Frequency per million instances` is computed. Therefore over all keys of the dictionary is looped, and the frequency is added to the NGrams instance in the dictionaries value.
The same is done for the bi- and trigrams, with the difference that instead of a single token a set of two or three tokens is used. These are joined together, with a space character inbetween, to form the identifier.

4.1.3 Autocompletion

The main "magic", of the code, happens in the `Autocompletion` class.
The user can type into the textBox. The event `_TextChanged` detects if the content in the textBox is changed. If this is the case, the content is taken as a string an in a first step converter to lower case characters, this since the tokenizer handles the tokens the same way. This results in fewer unique uni-, bi- and trigrams and therefore more data precise the code can work with. From this string the last to words are cut off, and used to find a fitting tri- or bigram, the latter only if no trigram is found. If only one word is in the string, a dot '.' is added in front of it.
When the user hits the `TAB` key, he receives the "auto-completion". To make it as similar as possible to a smartphone keyboard with suggestions, the user recieves the suggestion already when typing a word. As an example "The fir" leads to the following suggestions, sorted by their probability (Frequency per million instances, descending order):

1. st time
2. st place
3. st to
4. st of
5. e and

The first four suggestions, lead to the word "first" where as the fifth suggestion assumes "fire". If this feature is not wished, it can be deactivated by setting the constant `SUGGEST_TYPINGWORD`, in the Autocomplection class, to `false`.

4.2 Data

As the training (reference) data, a set of books from the Gutenberg Project is used. Some linguistic works and some scientific books. Also the *Bible* is part of the data set [1]. A detailed list of the books used can be found in the end of this document 12.
The number of tokens, and n-grams is found in Table 10 below:

# Tokens	5'250'734
# Unigrams	86'714
# Bigrams	1'168'147
# Trigrams	3'207'421

Table 10: Properties of the dataset

In Table **??** below, the 5 most common Uni, Bi and Trigrams are shown:

Most Likely Unigrams		**Most Likely Bigrams**		**Most Likely Trigrams**	
Unigram	FPMI	Bigram	FPMI	Trigram	FPMI
the	50876	of the	6651	. and the	365
.	49836	in the	4099	. it is	329
and	32774	. the	2953	of the lord	315
of	29508	. i	2765	the son of	272
to	22812	. and	2698	out of the	261

Table 11: Most likely N-grams and Frequency per million instances values

4.3 Instruction for using software

To use the above described software, follow the steps:

1. Load the data file (menu strip)
2. Tokenize the data by pressing the `Tokenize` button
3. Generate the Uni-, Bi- and Trigrams by pressing the `Gen. NGrams` button

   - The TextBox in the top can be used to type in

   - The ListBox shows the suggestions, in descending order (Frequency per million instances), if there are suggestions available
4. Start type text in the TextBox and accept the top suggestion by pressing the `TAB` key

[1] Tokenizing the dataset and creating the n-gram dictionaries is time consuming. On the authors computer it took 20 seconds for tokenization and 1 minute for generation of the 3 n-grams

4.4 Discussion

The results of this exercise are difficult to classify quantitatively. Therefore it's best for the reader to test it on it's own. What can be said is that, obviously, with such a (comparably to state of the art chatbots) small dataset, from which the n-grams are taken, the suggestions given are fairly limited. When a user keeps pressing the `TAB` key and just adding the most probable suggestion to the sentence, most likely he will end up with some non sense. Nevertheless, for sentences that are similar to the ones in the training data, the suggestions in a longer run are fairly ok. In the current state it is therefore not suggested to use the model in a generative way, but surely as a typing aid.

List of Figures

List of Tables

Title	**Author**
Forest Trees of Illinois- How to Know Them	Henry H. Gibson
Hawaiian Flowers	Helen Rockwell Foster
Histology of medicinal plants	William C. Steere
Botany for Ladies	Jane Loudon
Studies of Trees	Jacob Joshua Levison
Fungi- Their Nature and Uses	Mordecai Cubitt Cooke
The Genetic Effects of Radiation	L.J. Stadler
A Check-List of the Birds of Idaho	M. Dale Arvey
Problems of Genetics	William Bateson
Darwin	Charles Darwin
Natural Wonders	Edwin Tenney Brewster
Mammals from Southeastern Alaska	Gerrit Smith Miller Jr.
Worlds Within Worlds- The Story of Nuclear Energy, Volume 1 (of 3)	Isaac Asimov
Worlds Within Worlds- The Story of Nuclear Energy, Volume 2 (of 3)	Isaac Asimov
Worlds Within Worlds- The Story of Nuclear Energy, Volume 3 (of 3)	Isaac Asimov
Curiosities of Light and Sight	A. G. M. Michell
Our Atomic World- The Story of Atomic Energy	Edward Arthur Milne
The Principle of Relativity	Albert Einstein
A Room with a View	E. M. Forster
Cranford	Elizabeth Gaskell
Frankenstein; Or, The Modern Prometheus	Mary Shelley
History of Tom Jones, a Foundling	Henry Fielding
Little Women; Or, Meg, Jo, Beth, and Amy	Louisa May Alcott
Middlemarch	George Eliot
Moby Dick; Or, The Whale	Herman Melville
Pride and Prejudice	Jane Austen
The Adventures of Ferdinand Count Fathom - Complete	Tobias Smollett
The Adventures of Roderick Random	Tobias Smollett
The Blue Castle- a novel	Lucy Maud Montgomery
The Complete Works of William Shakespeare	William Shakespeare
The Enchanted April	Elizabeth von Arnim
The Expedition of Humphry Clinker	Tobias Smollett
Twenty years after	Alexandre Dumas

Table 12: Table of books used for the dataset in Problem 4