Natuurlijke Taalmodellen & Interfaces
Tejaswini Deoskar
Part A (Step 4)

Lucas van Berkel, 10747958
Ruben Gerritse, 10760326
Gideon Mooijen, 10686290

Designing a POS-tagger, based on smoothing methods, representation as a Hidden Markov model and the Viterbi-algorithm.

March 1st, 2016

Abstract

Assigning POS-tags to an unannotated sequence of strings is a difficult task for a computer. With help of smoothing, emission probability, state transition probability, representing the path as a Hidden Markov model and finding a path in this representation through the Viterbi-algorithm, the program performs very well. Without smoothing 86.28% of the words are assigned the correct tag. With smoothing a precision of 85.70% is obtained. Considering the use of a finite training set with limited diversity, the performance can be regarded as successful.

1 Introduction

During the first half of this course, the goal was to gradually build a POS-tagger. In this program, the concepts discussed in class - *smoothing, Viterbi-algorithm* - must be included. To obtain this very program, (parts of) previous code formed the base for each next step. In this report there are often references to either one of those three steps.

1.1 Hidden Markov model

A *'Hidden Markov model'* (HMM) is a statistical model which uses hidden nodes besides the Markov system. One way to make a proper *'Part-Of-Speech tagger'*, is to use the probability of a tag given a certain word, but in that case the same tag will always be assigned to the given word. In order to make the model more flexible, one could calculate the next tag given the previous tag, hence the hidden layer. Since a Markov model uses the *'Bayes'* rule as a standard, the same accounts for the HMM. To calculate the maximum probability of a chain of tags given a chain(*sentence*), the probability of the sentence given the possible tags and the probability of the tags themselves has to be calculated. This results in the following equation:

$$P(T \mid S) = \frac{P(S \mid T)\, P(T)}{P(S)} \tag{1}$$

Since the probability of sentence S itself has no influence on the chain of tags with the maximum probability, it is left out. The two probabilities that have to be calculated are called the *'emission probability'* and the *'state transition probability'*. The emission probability takes a given tag and returns the probability of the word given the tag. This is calculated through the following equation:

$$P(w \mid t) = \frac{Count(w_t)}{Count(t)} \tag{2}$$

The state transition probability is the hidden layer in the model. To calculate the probability, the frequency of bigrams and unigrams is counted of POS-tags only. The resulting equation is the following:

$$P(t_i \mid t_{i-1}) = \frac{Count(t_{i-1}\, t_i)}{Count(t_{i-1})} \tag{3}$$

By combining the two equations the probability of the POS-tags of the sentence can be calculated.

1.2 Viterbi-algorithm

The Viterbi-algorithm is a way to obtain a hidden *'Viterbi-path'*, which is the most probable cause of the observed events. In terms of POS-tagging, the observed events are the words that require a tag, the Viterbi-path is the most probable sequence of tags. This algorithm is an example of *'dynamic programming.'* The main reason that this type of programming has acquired a special name, is the fact that programs of this type are computationally more powerful than non-dynamic programs. Although the goal remains the same (finding the optimal path), dynamic programs do not regard the problem as one large task, but a set of smaller problems. Then, calculations of the outcome of those *sub-problems'* are stored. This step is called *memorization'*, storing the outcomes of sub-problems that needs to be used later on, rather than computing them again.

1.3 Using previous work

Since the POS-tagger is built with pieces of code made earlier in the process, a small overview of the influence of every part in this final assignment is included in the section below.

1.3.1 Step 1: Extracting n-gram statistics.

In the first step, a dictionary was used as a data structure to keep track of the sequences contained in a test corpus. A dictionary was used because of its ability to link a key and a value. For the calculation of the probabilities of a sentence, not only the sequences themselves need to be tracked but also the number of occurrences of the sequences in that sentence.

1.3.2 Step 2: Markov language models using n-gram statistics

The second program contained a language model to calculate the frequency of the n-grams in a certain corpus. From the frequencies the posterior probability could be calculated. The same principle is used for the fourth program, only the sequence of tags is counted and used for the state transition probability.

1.3.3 Step 3: Smoothing n-grams statistics

For natural language purposes - *regarding calculating probabilities* - smoothing is used to make a more realistic view of the occurrences of sentences in a language. Without smoothing, the values of the calculated probabilities will result in a skewed view on possible sentences. N-grams that are assigned a value 0 - *because they do not occur in the corpus* - will be regarded as an impossible combinations of words, rather than an unlikely combination of words. Due to the distinct computational ceiling it is impossible to calculate all exact probabilities of n-grams. To solve this problem, we apply smoothing to be able to make more realistic assumptions about these probabilities. This results in better generalizations and thus we obtain a better view of the language and the occurring n-grams, even though the corpus we used is finite.

1.4 Research question

What is the amount of precision that a POS-tagger, including the techniques of Hidden Markov models, smoothing and the Viterbi-algorithm, can achieve?

2 Method

2.1 Extracting data from corpus

The first step to retrieve the data from the corpus is by creating a useable representation of the data. This has been achieved by reading the whole text-file and splitting it into sentences when either a full stop is read or a line of ='s

. Each sentence will be converted to a list of the words in that sentence. The second step is to count occurrences of unigrams and bigrams. The obtained sentences consist of word-tag pairs in the form *"word/tag"*. Using this data, three dictionaries need to be constructed for the tagging of POS-tags. The first constructed dictionary *'tag_dict'* contains the unigrams of all the tags occurring in the corpus. The second dictionary *'wordTag_dict'* contains all word-tag combination in the corpus. The third dictionary *'tagSeq_dict'* contains all the occurring tag bigrams. Also the number of occurrences of all those keys in the dictionaries will be tracked as the values in the dictionaries.

2.2 Calculating probabilities

To calculate the probability of a tag sequence, the Viterbi-algorithm is implemented. Starting with a dictionary containing only the *"START"* tag, each key in the dictionary will be expanded with all the possible tags for the next state. For each tag, only the sequence with the highest probability ending with that tag will be saved to the dictionary. This will be repeated until the *"STOP"* tag. At the end only the tag sequence with the highest probability will be contained in the dictionary.

2.3 Smoothing n-gram statistics

In the program there is also the option to smooth the dictionary before calculating the probabilities. The three dictionaries that are used are smoothed differently. The *'wordTag_dict'* dictionary is smoothed by giving half of the counts of *'wordTag_dict'* combinations with count one to the unknown word. That is if you assume that there is only one unknown word in the test corpus, which was advised by the teaching assistants. This method was used to make unseen words probable. Another way to smooth low-frequency counts is the *'Good-Turing method'*. The GT-method is a method that "takes from the rich and gives to the poor".

$$r^* = \frac{(r+1)\frac{n_{r+1}}{n_r} - r\frac{(k+1)n_{k+1}}{n_1}}{1 - \frac{(k+1)n_{k+1}}{n_1}} \tag{4}$$

For our assignment we smoothed the counts from 1 to 4 of the tag-sequence dictionary, using the GT-method. From the updated dictionary the new tag dictionary was formed, adding the counts of the tags individually.

2.4 Handling unseen words

There are two different approaches to the test corpus, unsmoothed and smoothed. Each way gives a different approach to unseen words. The unsmoothed dictionary assigns the unseen words a tag called 'X', an imprecise method because it assigns every future probability of the sentence zero *('X' is an nonexistent tag)*. This results in every future tag to be randomly generated. The smoothed dictionary is a more sophisticated method, it calculates the emission probability through the next equation for an unseen word w:

$$P(w \mid t) = \frac{1}{2}\frac{n_1(t)}{N(t)} \tag{5}$$

The state transition probability can not receive an unseen tag, but it is not possible to assign a tag to a word that does not exists.

3 Results

Method	**No smoothing**	**Smoothing**
Amount correct	6977	6930
Percentage correct	86.28 %	85.70 %

4 Discussion

4.1 Difference in accuracy

The difference in accuracy is unexpected. The smoothed dictionaries gave a result which was worse than the unsmoothed dictionaries. After investigating the wrong tags it can be concluded that most of the wrong tags were a result of sentences containing unseen words. Where the unsmoothed dictionaries assigned them with tag 'X', the smoothed dictionaries attempted to assign a valid tag. Assigning the tag 'X' would result in a state transition probability of zero, so the rest of the tags of the sentence were assigned as unigrams. But assigning the wrong but existing tag to the unknown word has proved not to improve the result and even slightly lower the accuracy.

4.2 Conclusion

Within the field of natural language there is a great diversity regarding word/tags-combinations. It is safe to assume that not all of these combinations occur in the training set, and when the program analyzes the test set it will encounter combinations that it is not familiar with. There can also be words with multiple tags, sentences with tag-sequences that it is not familiar with. There might even be words in the test set the program has never seen before. In short, the amount of natural language covered in the training set is negligible, and the amount of possible ambiguity due to all reasons mentioned above is very high. With that in mind, the performance of the program is very well, considering the fundamentals of the program are assumptions. Yet, the fact that the smoothing did not result in higher accuracy was not in line with our expectations.

4.3 Problems regarding extraction from the corpus

Upon finalizing the code and the report, a difficulty was encountered regarding the distillation of the words and their corresponding tags. The assumption was made that all tags were presented in the form the/DT or $role/NN$ (of all tags that do not contain punctuation marks). This appeared to not be the case, as sequences of the form $male/JJ|NN$ or $x/y/TAG$. Due to lack of time it was resolved by eliminating the last tag (in the case of multiple tags) or eliminating every word except for the last (in the case of multiple tags).

4.4 Further improvements

There are a few possibilities to increase the accuracy of the POS-tagger. Three of these have been examined.

1. Increase training set
2. Increase the size of the n-grams
3. Include word-specific sub-trees

All of these will increase the performance of the POS-tagger[1].

5 How to run

```
BerkelGerritseMooijen4.py -smoothing [yes|no] -train-set [path]
-test-set [path] -test-set-predicted [path]
```

[1] Hirshman, B. R., (2006). *Training Set Properties and Decision-Tree Taggers: A Closer Look*, Unpublished manuscript, Carnegie Mellon University, Pittsburgh, PA