Final Paper: Language models BSc AI 2013 Assignment 4

Eszter Fodor, Sharon Gieske
(5873320, 6167667)

May 23, 2013

Abstract

This paper discribes our implementation of a probabilistic Markov POS tagger, trained and tested on various sections of the Wall Street Journal. The POS tagger is trained by creating a language and task model, trained with smoothing and without smoothing. The test corpus is tagged using the Viterbi algorithm. A POS tagger trained with smoothed models shows to have a good accuracy and recall.

Introduction

The problem that needed to be solved during the four assignments was how to calculate the most probable Part-Of-Speach (POS) tag sequence using tagging and smoothing on a training corpus. The program that needed to be written would eventually be able to tag the test corpus with POS tags.

Some POS-tag sequences are less possible than others, this information can be used to analyze texts to learn whether it is grammatically correct or not. Another importance of POS-tagging is to disambiguate words within a sentence. This can be done by taking the previous words and their POS-tags into account. A program that is able to tag sentences accurately can be used on various corpora to learn what kind of sentences it consists of.

Approach

This section describes the steps that were taken to construct a working program that is able to POS-tag a corpus as accurately as possible.

Basics

There are various models which form the base of a working POS-tagger. These models need to be implemented in order to successfully build a tagger. The implementation of these models was the objective of the first three assignments of the course Natural Language Models. A small portion of the functions within the implemented programs were not used to construct the tagger and will therefore not be discussed in this paper.

The remaining implemented functions, such as constructing *ngrams*, implementation of *Good-Turing smoothing*, implementation of the *Viterbi algorithm* will be discussed further along.

N-grams

N-grams are sequences of n items from a sequence of text. N-grams are constructed using tuples of n items. A dictionary is created using these n-grams to obtain their frequency and is used for calculating sequence probabilities.

Good-Turing smoothing

Good-Turing smoothing is a technique to predict the probability of occurances of items not seen in training. Good-Turing uses the count of things seen once to estimate the count of unseen items using the following equation:

$$r_0 = \frac{n_1}{n_0}, \text{ for } r = 0 \tag{1}$$

Combining Good-Turing smoothing with Katz-Backoff allows discounting over events occuring $1 \leq r \leq k$ times.

$$r^* = \frac{(r+1)\frac{n_{r+1}}{n_r} - r\frac{(k+1)n_{k+1}}{n_1}}{1 - \frac{(k+1)n_{k+1}}{n_1}}, \text{ for } 1 \leq r \leq k \tag{2}$$

Training

Sections 2 to 21 from the Wall Street Journal formed the training corpus for this project. Every word in this corpus has been previously POS-tagged, taking the context within the sentence into account. These words, along with their tags, were loaded as tuples in the form of $(word, tag)$ into the program to be used later on. Besides the tuples for word-tag pairs, a dictionary with unigrams of every word in the corpus along with their frequencies is created. Furthermore, trigrams are created from all the POS-tag sequences within the corpus along with their frequencies, which forms a so called *language model*.

Another model that needed to be implemented is the so called *lexical model*, which is named *task model* in the code to prevent confusion while programming. This model depends on the the words and the tags that occur in the corpus and on the dictionary with POS-tag unigrams mentioned above. Using these three acquired data the lexical model is created by calculating the probability of the word-tag pair. This probability is gained by counting the occurances of the tag given by that word divided by the total occurances of this tag. Furthermore, the possibilities found for POS-tags for a word are being kept as well as the total occurances of POS-tags.

When running the program smoothing is by default enabled which means that the two models are smoothed, though both in different ways. The *language model* is smoothed using Good-Turing smoothing with Kats Backoff with $k = 4$. In the case of the lexical model, only events with frequencies of 0 and 1 are smoothed using the formulas:

$$0^* = \tfrac{1}{2} \cdot \tfrac{n_1(t)}{n_0} \text{ and } 1^* = \tfrac{1}{2}$$

The tagger is trained, using all of the above, by calculating the probabilities for the POS-tag sequences. Unfortunatly the training takes a relatively long time because the calculations for the lexical model take much data into account and it takes time to process it all, especially with a corpus that is as big as the sections of the Wall Street Journal.

Tagging

The next step, after the training of the tagger is tagging of the test corpus. This corpus consists of the sentences from section 23 of the Wall Street Journal. The tagging of this corpus is done by using the *Viterbi algorithm*. This algorithm uses *emission probabilities* and *transition probabilities* to calculate the most probable tag sequencies.

Emission probabilty the probability of a word, given the current state (the POS-tag in this case). This probability is calculated by applying the lexical model to every word and their tags. Transition probability is the probability of a state given the previous states and in this case the probability of a POS-tag, given the previous two POS-tags within the sentence.

After calculating these probabilities, the most probable route between states is calculated by recursively calculating the most probable transition between states. How the algorithm works is demonstrated in the appendix by *Algorithm 2* and *Algorithm 3*.

The words from the test corpus, along with their POS-tags that were calculated by the steps described above, are written in an output file. To check whether the tagged did a good job, the output file needs to be compared with the provided test corpus and the tags that were manually added to it. Precision and recall are calculated to determine the quality of the POS tagger.

Testing

The program can be run by:

```
$ python taalmodellen4.py training.pos test.pos output.pos
```

With this command the program is fully run including smoothing, tagging and comparing the resulting tags to the tags in test corpus. The output that is created is an output file with the words from the test corpus and their tags and in the console the *precision* and the *recall* of the tagger is printed along with the time that it took the program to finish. The results of the program can be found in section *Results*.

Results

The results of the final program with smoothing enabled:

```
sharon@sharon-Aspire-5749 ~/Documents/uva/taalmodellen/taalmodellen/4 $ python
taalmodellen4.py training.pos test.pos output.pos
> Smoothing enabled
> Write output to bla.pos
** Loading train corpus
** Loading test corpus
** Calculating unigram
** Calculating language model
** Calculating task model
*** Processing 42916 sentences
** Smooth language model
** Smooth task model
** Calculate special bigram
** Start Tagging
-----------------------
Total words: 49443
Total words tagged: 48103
Precision: 80.342182%
Recall: 78.164755%
Time taken: 57 min and 3 sec
```

The results of the final program with smoothing disabled:

```
eszter@eszter-laptop /media/DATA/AI/taalmodellen/4 $ python taalmodellen4.py
training.pos test.pos nonSmoothTest.pos --no-smoothing
> Smoothing disabled
> Write output to nonSmoothTest.pos
** Loading train corpus
** Loading test corpus
** Calculating unigram
** Calculating language model
** Calculating task model
*** Processing 42916 sentences
** Start Tagging
-----------------------
Total words: 47066
Total words tagged: 45821
Precision: 55.956876%
Recall: 54.476692%
Time taken: 40 min and 31 sec
```

Conclusion/Discussion

There is a big difference between the precision and recall of the tagger when it is run with and without smoothing. As it can be read above, the precision of the tagger *with* smoothing is 80% and *without* smoothing it is only 56%. This enormous difference can be explained by the fact that POS-tags can occure with a very low frequency which makes it harder to calculate the highest probability for a sequence.

Though this tagger is not perfectly accurate, it does a decent job. It could deffinetly use some tweaking if one wanted to use it for research purposes, but as of this project a precision and recall of around 80% is satisfactory.

The POS tagger with smoothing gives a very good result on the test corpus, but this does not mean the POS tagger is directly suitable for texts from other domains. The POS tagger is wholely trained on text obtained from the Wall Street Journal. Word-tag pairs are trained domain specific and it is not certain the POS tagger will perform as well on texts from other domains.

A Algorithms

Algorithm 1 Viterbi algorithm

Input: sentence: sentence to be tagged
Input: langugageModel: second-order Markov model of POS-tags
Input: bigran: first-order Markov model of POS-tags
Input: taskModel: lexical model
Input: wordTags: mapping from word → tags
V ← ∅
forall the *word* ∈ *sentence* **do**
  emission ← ∅
  forall the *tag* ∈ *wordTags* **do**
    emission(tag) → taskModel(word, tag)
  end
  V ← V ∪ emission
end
transition ← ∅
for $i = 1 \rightarrow \|V\|$ **do**
  forall the $e \in V_i$ **do**
    forall the $f \in V_{i-1}$ **do**
      forall the $g \in V_{i+1}$ **do**
        if $\langle e, f, g \rangle \in langugageModel \wedge \langle e, f \rangle \in bigram$ **then**
          transition(f, e) → $\frac{languageModel(e,f,g)}{bigram(\langle e,f \rangle)}$
        else
          $transitions(f, e) \rightarrow 0$
        end
      end
    end
  end
end
return *calculateOptimalRoute(V, transition,* $\|sentence\|$*)*

Algorithm 2 calculateOptimalRoute

Input: V: a state trellis of a HMM
Input: transition: transition probabilities of a HMM
Input: i: current position in trellis
if *location* == −1 **then**
  return (1, START)
else
  $\langle prob_prev, path_prev \rangle \leftarrow calculateOptimalRoute(V, transitions, i-1$
  $\langle prob, tag \rangle \leftarrow argmax(prob = transition(path_prev(i-1), tag | tag \in V_i)$
  return $\langle prob_prev \cdot path \cup tag \rangle$
end