HMM tagger for Swedish

Gordana Ilić Holen

May 15, 2009

1 Introduction

I will start this paper with a general introduction to HMMs and the decoding challenge.

Due to atrophy of my programming muscles I have chosen to use TnT tagger (Brants 2000) on the Stockholm-Umeå Corpus (SUC) data. I will use rest of the paper to describe TnT's tagging method and results, its smoothing technique, and I will conclude with a short evaluation of its results. While working on this assignment, I was using Manning and Shütze's Foundations of Statistical NLP, and Jurafsky and Martin's textbook on Speech and Language Processing (Jurafsky and Martin 2008; Manning and Schütze 1999).

2 HMM - an introduction

Hidden Markov Model is a probabilistic function defined by the following parameters:

$Q = q_1 q_2 ... q_N$ A set of N states.

$A = a_{11} a_{12} ... a_{nn}$ A transition probability matrix, where each a_{ij} represents a probability of transition from state i to state j. The sum of all the transitions from any state i to any state j is 1.

$O = o_1, o_2 ... o_T$ A set of T observations.

$B = b_i(o_T)$ A sequence of emission probabilities, expressing the probability of a state i generating (emitting) an observation o_t.

q_0, q_F A special start state q_0 and a special final state q_F that do not emit any observations. Sometimes instead of distinct start states, all states have start probabilities, the sum of which is 1.

The illustration in Fig. 1 represents a simple form of HMMs, called Markov chain. The red circles represent the hidden states, and the green circles represent observations. The red arrows represent transition probabilities and the green ones emission probabilities. The blue circles are the start and end states, that do not have emissions, and the blue arrows represent the transmission probabilities with value 1.0 from the start state and to the end state.

When applied to tagging, the visible states are words, and the invisible are tags. Tagging can be defined as finding the most probable sequence of tags,

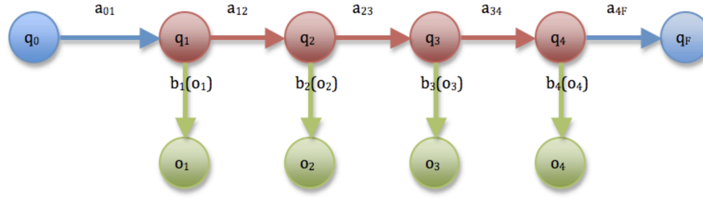

Figure 1: A model of a Markov chain

given the sequence of words. Or, more formally, given the observation sequence O, and the model μ, choosing a state sequence $(q_1...q_T)$ that best explains the observations. This is known as the *decoding problem*.

We start from wanting to find an estimate of the best string of tags from *1* to n ($\hat{t}_1^n$), given the string of words w_1^n:

$$\hat{t}_1^n = \operatorname*{argmax}_{t_1^n} P(t_1^n|w_1^n) \tag{1}$$

Applying Bayesian inference (2) to our estimate gives us the equation (3):

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{2}$$

$$\hat{t}_1^n = \operatorname*{argmax}_{t_1^n} \frac{P(w_1^n|t_1^n)P(t_1^n)}{P(w_1^n)} \tag{3}$$

But since $P(w_1^n)$ in this case is the string of words that we are trying to find the best tag sequence for, and this string of words is always the same, we can drop the denominator and reduce the equation (3) to (4):

$$\hat{t}_1^n = \operatorname*{argmax}_{t_1^n} P(w_1^n|t_1^n)P(t_1^n) \tag{4}$$

We now apply some simplifications based on general assumptions about the nature of HMMs:

- We assume that the words tag only depends on the previous tag (The limited horizon assumption)
- and that this dependency does not change over time (The time invariance assumption)

The simplification of the equation (4) based on the limited horizon assumption yields equation (6) for the second multiplier:

$$P(t_i) \approx \prod_{i=1}^{n} P(t_i|t_{i-1}) \tag{5}$$

In addition to the limited horizon assumption, we make two assumptions about the words:

- Words are independent of each other.
- Words identity depends only on its tag.

Thus we can reduce the first multiplier of the equation (4) to equation (6):

$$P(w_1^n|t_1^n) \approx \prod_{i=1}^{n} P(w_i|t_i) \tag{6}$$

Simplifications (5) and (6) put together with equation (4), give:

$$\hat{t_i^n} \approx \underset{t_i^n}{\text{argmax}} \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1}) \tag{7}$$

And we have now finally reduced the problem to a level where we can compute values by using the data from the corpus. The part of the equation that computes the possibility of seeing a tag given its preceding tag is computed by counting (C) all the instances of the tag t_i appearing before the tag t_{i+1}, and dividing the number with the number of all observations of the tag t_i:

$$P(t_i|t_{1+1}) = \frac{C(t_i, t_{i+1})}{C(t_i)} \tag{8}$$

Similarly, the probability of a word w_i getting a tag t_i is computed by counting all the instances of word w_i which have gotten tag t_i in the training corpus and dividing them with the total number instances of word w_i in the corpus.:

$$P(w_i|t_i) = \frac{C(w_i, t_i)}{C(w_i)} \tag{9}$$

The Viterbi algorithm is an implementation of the equation (7):

$$v_t(j) = max_{i=1}^{n} v_{t-1}(i) a_{ij} b_j(o_t) \tag{10}$$

In (10), the max operator keeps track of the previous most probable tag, $v_{t-1}(i)$ is the Viterbi value for the previous algorithm step, a_{ij} is transition probability from the state q_i to current state q_j, while $b_j(o_t)$ is observation likelihood of symbol o_t given the state j. The a_{ij} part of the formula is representing the equation (8), while $b_j(o_t)$ is covering the equation (9). Those two types of probabilities are known as respectively *lexical probabilities (8)* and *contextual probabilities (9).*

3 TnT tagging

I used a tiny Pearl script to format the training file to a form expected by TnT. Thus formatted file contains one word token and its tag in each line, and was saved as `suk.tt`. It was used for training the tagger and this was done by running the `tnt-para` script on the `suk.tt` file. This resulted in two files: `suk.123` and `suk.lex`. The file `suk.123` contains the tag n-grams (uni-, bi- and trigrams) from the `suk.tt` corpus, i.e. the model's contextual probabilities, while the `suk.lex` contains the lexical probabilities.

According to the suk.123 file, the suk.tt corpus contains 24 unigrams, 383 bigrams and 3204 trigrams. The 24 unigrams correspond to the SUC corpus' 23 tags plus one start state __$.

We can, for instance, look at one of the smallest tags, HS (Wh-possessive), and see the following:

```
HS      8
        KN      1
                PS      1
        NN      4
                AB      2
                VB      1
                PN      1
        JJ       3
                NN      3
```

The tag HS appeared 8 times, once followed by tags KN PS, four times followed by tag NN, and of those four times, twice the tags HS NN were followed by AB, once by VB and once by PN. And finally, we have had three occurrences of trigram HS JJ NN.

It seems that TnT has treated the delimiter tag DL like any other tag, so that the whole file was treated like one sentence. For that reason, the start state __$. has been used only twice[1], one of these times obviously at the start of the file, right before the PM and KN tag (The first two word/tag pairs in the training corpus are Per PM and och KN).

```
    __$     2

            PM      1
                    KN      1
```

The suk.lex contains information that the corpus contains 101961 word-tokens, and 19555 distinct word types. It lists the word types alphabetically, together with the information on how many times the word token appeared in the corpus, and how many times with each particular tag:

```
Den                     164     DT      113     PN      51
Denkert                 4       PM      4
```

This extract from the suk.lex file shows that the word Den occurred 164 times in the test corpus, and that it had 113 times the determiner tag DT, and the pronoun tag PN rest of the times, while the word Denkert occurred four times, and was always tagged with the proper name tag PM.

These two files, suk.123 and suk.lex contain all the information needed to apply the Viterbi algorithm to a tag a new corpus. That is done by running the command tag, using suk.123 and suk.lex files as a model and file suk.tt as the untagged corpus. The suk.tt file is generated by running another tiny

[1] I am not sure why it is written that the __$ tag occurred twice, in all the other tags I have checked, the number of the unigrams was – as we would expect – a sum of both the number of bigrams and the number of trigrams, just like in the HS tag example.

Pearl script to the provided SUC test corpus, which stripped it to a form where each line contains only the word token and nothing else. The `tnt` command yields the file `suk.tts`, which is an instance of our test corpus tagged by TnT.

3.1 Smoothing

I have earlier in this paper shown how we came to the equation (7) which made a basis for the Viterbi algorithm. This equation concerns however the first-order HMMs, the ones we were supposed to implement ourselves for this assignment. I have however used a ready implementation which was based on a second-order HMM. To extend this formula to the trigram model, the limited horizon assumption has to be extended to *"We assume that the words tag only depends on the **two** previous tags"*, and our (4) equation extends thus to (11):

$$\hat{t}_1^n \approx \underset{t_i^n}{\operatorname{argmax}} \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1}, t_{i-2}) \tag{11}$$

This poses the problem of data sparsity, which is much more serious then a first-order HMM would have encountered. For that reason, we cannot get the maximum likelihood estimation by just extending the (8) formula to (12), as it is not improbable that the trigram was not seen before:

$$P(t_i|t_{i-1}t_{i-2}) = \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})} \tag{12}$$

To avoid piling-up of zero-probabilities we introduce weighted sum of uni-, bi- and trigram probabilities instead of simple counts :

$$P(t_i|t_{i-1}t_{i-2}) = \lambda_1 \hat{P}(t_i) + \lambda_2 \hat{P}(t_i|t_{i-1}) + \lambda_3 \hat{P}(t_i|t_{i-1}, t_{i-2}) \tag{13}$$

This smoothing technique is known as *linear interpolation.* The λs are computed in the following way: for each trigram t_1, t_2, t_3 were computed the following values:

$$C_1 = \frac{C(t_1, t_2, t_3) - 1}{C(t_1, t_2) - 1} \tag{14}$$

$$C_2 = \frac{C(t_2, t_3) - 1}{C(t_2) - 1} \tag{15}$$

$$C_2 = \frac{C(t_3) - 1}{N - 1} \tag{16}$$

Then, if (14) was highest, λ_3 was incremented by C_1, if C_2 was the highest λ_2 was incremented by C_2, and if C_3 the was highest, λ_1 was incremented by the count. The values of λs TnT has come to are: $\lambda_1 = 0.2020891$, $\lambda_2 = 0.2961259$ and $\lambda_3 = 0.5017850$, and $\lambda_1 + \lambda_2 + \lambda_3 = 1$, as required.

3.2 Evaluation

TnT offers a script (`tnt-diff`) to compare the statistically tagged corpus (`suk.tts`) with a correct corpus. The result of 90.5% was far from impressive when compared to the 96.7% the tagger achieved when trained on PennTreebank, which is as good as a tagger gets today.

TnT also offers a choice to run the tagger after uni- or bigram model. Initially, I found the results confusing:

```
Overall result (trigram):
Equal  :   20574 /  22734 ( 90.50%)
Different:    2160 /  22734 (  9.50%)


Overall result (bigram):
Equal  :   20574 /  22734 ( 90.50%)
Different:    2160 /  22734 (  9.50%)


Overall result (unigram):
Equal  :   20572 /  22734 ( 90.49%)
Different:    2162 /  22734 (  9.51%)

19727 tokens known (86.77%), 3007 unknown (13.23%).
```

The results of a bigram model were identical to those of the trigram model, down to the last word. The only difference was, of course, λ values: $\lambda_{bi1} = 0.2553035$ $\lambda_{bi2} = 0.7446965$. Apart from the obvious $\lambda = 1$ value, the unigram model differed from bi- and trigram model in only one word: While the word *svårt* consistently got the adverb tag `AB` by the bi- and trigram tagger, it got the adjective tag `JJ` by the unigram tagger. However, none of them was right, as the test corpus shows the word tagged twice with `JJ` and four times with `AB`.

I think that due to the exceptionally small training corpus, the tagger virtually did not encounter bi- and trigrams it had learned from the training corpus and had to base its calculations on unigrams. As the unigram model offers a much weaker description of language then bi- and trigram models do, the comparatively bad results do not come as a surprise.

References

Brants, T. (2000). Tnt – A statistical part-of-speech tagger. In *ANLP 2000*, Seattle, WA, pp. 224–231. URL: http://www.coli.uni-saarland.de/ thorsten/tnt/.

Jurafsky, D. and J. H. Martin (2008). *Speech and Language Processing* (2nd edition ed.). New Jersey: Pearson Prentice Hall.

Manning, C. D. and H. Schütze (1999). *Foundations of Statistical Natural Language Processing*. Cambridge, Massachusetts: The MIT press.