Spaghetti and HMMeatballs

Cooking a low-fat statistical tagger in Prolog

Torbjörn Lager
Department of Linguistics
University of Göteborg
`Torbjorn.Lager@ling.gu.se`

Abstract

Typically, a statistical part of speech tagger uses the *Viterbi algorithm* to find the most probable path through a *Hidden Markov Model* (HMM). In this short paper, a very small and simple such tagger is developed in the Prolog programming language. The major motivation behind the implementation is pedagogical, but the result is also remarkably easy to use for practical purposes.

1 Introduction

Typically, a statistics-based method for automatic part-of-speech tagging represents the required knowledge as a probabilistic model. Most popular among such models is the *Hidden Markov Model* (HMM). In a HMM there are *states*, *transitions* between states, and *symbols* emitted by the states. There are two kinds of probabilities associated with a HMM: *transition probabilities*; i.e. the probability of a transition from one state to another, and *output probabilities*, i.e. the probability of a certain state emitting a certain symbol. The model is called *hidden* since from watching the string of symbols that it outputs, we cannot in general determine which states it passes through.

I will illustrate here with a simple HMM in which the states represent parts-of-speech, and the symbols emitted by the states are words. The assumption is that a word depends probabilistically on just its own part-of-speech (i.e. its tag) which in turn depends on the part-of-speech of the preceding word (or on the *start* state in case there is no preceding word).[1]

Our sample HMM can be displayed as in Figure 1.

[1]. This is the assumption of a so called *biclass* model. It is more common in fact to use a *triclass* model, i.e. to assume that the part-of-speech of a word depends on the parts-of-speech of the preceding *two* words.

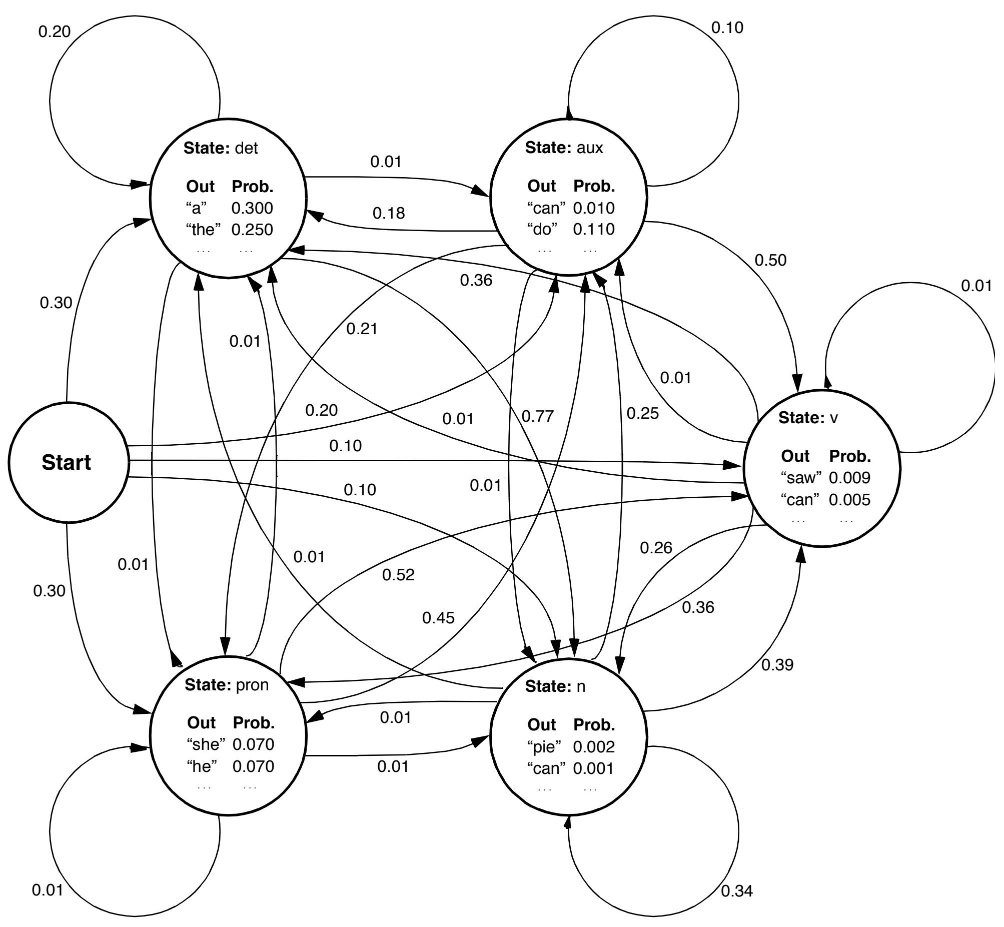

FIGURE 1. A Hidden Markov Model

Needless to say, in a realistic application, the HMM will have to be very much bigger. Still, this particular HMM is capable of generating, among other strings, the sentence "he can can a can". It passes from the *start* state to the state *pron*, from there to *aux*, then to *v*, and from *v* to *det* and finally to *n*, and outputs the words as it moves from state to state. The probability of this happening is easy to compute: we simply multiply all the transition probabilities and output probabilities along the path. This particular HMM could generate the same string of words by going through other states (e.g. start-pron-aux-aux-det-aux) but that would be less probable.

The above, however, is a good description of string *generation*, but not of part-of-speech *tagging*. In part-of-speech tagging, the string of symbols is given and the task is to find the sequence of state transitions most likely to have generated this string.

2 A Naive Algorithm

To tag a given text, we need a program that computes the sequence of state transitions that has the highest probability of being the one that generated the text. Let us assume that the output probabilities are represented as clauses of the form *outprob(Word, Tag, Probability)*, e.g. as:

```
(1)     outprob(a,det,0.300).
        outprob(can,aux,0.010).
        outprob(can,v,0.005).
        outprob(can,n,0.001).
        outprob(he,pron,0.070).
```

Furthermore, let us assume that the transitional probabilities are represented as clauses of the form *transprob(Tag1,Tag2, Probability)*, e.g. as:

```
(2)     transprob(start,det,0.30).        transprob(v,det,0.36).
        transprob(start,aux,0.20).        transprob(v,aux,0.01).
        transprob(start,v,0.10).          transprob(v,v,0.01).
        transprob(start,n,0.10).          transprob(v,n,0.26).
        transprob(start,pron,0.30).       transprob(v,pron,0.36).

        transprob(det,det,0.20).          transprob(n,det,0.01).
        transprob(det,aux,0.01).          transprob(n,aux,0.25).
        transprob(det,v,0.01).            transprob(n,v,0.39).
        transprob(det,n,0.77).            transprob(n,n,0.34).
        transprob(det,pron,0.01).         transprob(n,pron,0.01).

        transprob(aux,det,0.18).          transprob(pron,det,0.01).
        transprob(aux,aux,0.10).          transprob(pron,aux,0.45).
        transprob(aux,v,0.50).            transprob(pron,v,0.52).
        transprob(aux,n,0.01).            transprob(pron,n,0.01).
        transprob(aux,pron,0.21)          transprob(pron,pron,0.01).
```

In principle, the simple Prolog program in (3) would do just fine.

```
(3)     most_probable_sequence(Words,Ss) :-
              findall(PS,sequence(Words,1-[start],PS),PSs),
              max_key(PSs,P-Ss1),
              reverse(Ss1,[start|Ss]).

        sequence([],PSs,PSs).
        sequence([Word|Words],P1-[S1|Ss],PSs) :-
              outprob(Word,S2,Po),
              transprob(S1,S2,Pt),
              P2 is Po*Pt*P1,
              sequence(Words,P2-[S2,S1|Ss],PSs).
```

Here are some implementation details: The set of probability-sequence pairs are represented as lists of terms of the form *P-Sequence*, where *Sequence* is a list of terms representing tags, and *P* is a number representing the probability for that sequence. During processing, it is convenient to keep the sequences in reverse order (thus P_i-$<T_1,..,T_{i-1},T_i>$ is represented by the term P_i-$[T_i,T_{i-1},..,T_1]$) so that adding a tag to a sequence becomes a matter of putting it first in the list. Note also the use of a dummy 'start' element. This is so that we do not need to complicate pattern matching further down in the program. The predicate *max_key/2* is a simple utility predicate that finds the element with the largest key (i.e. the maximum probability sequence in our case) in a list of key-value pairs. The last thing this program does is to reverse the most probable sequence and remove the dummy.

Given our sample HMM, the most probable sequence of tags corresponding to "he can can a can" is computed as follows:

```
(4)     ?- most_probable_sequence([he,can,can,a,can],Sequence).
        Sequence = [pron,aux,v,det,n]
```

This program is horrendously inefficient, however, since it examines *every* sequence of states that could possibly generate the text, which indeed can be very many. In our HMM example there are 'only' 27 different sequences of states that generate "he can can a can" but that sentence is very short and with longer sentences and bigger HMMs we are looking at thousands or even hundreds of thousands of possible sequences. Hence, a naive algorithm will not do.

3 The Viterbi Algorithm

Fortunately, there are algorithms that will find the most probable sequences very fast, without having to calculate the probabilities for all sequences. Here a Prolog implementation of one such algorithm – the *Viterbi algorithm* (Viterbi 1967) – will be given.

As before, the goal is to map a sequence of words $<W_1,..,W_n>$ into the most probable sequence of tags $<T_1,..,T_n>$. The basic idea is to maintain a set *PSs* of possible sequences and to discard the impossible ones as early as possible. Suppose we read a word W_i from the input list of words. Then for each state T_i in the HMM such that T_i outputs W_i, and for each sequence P_{i-1}-$<T_1,..,T_{i-1}>$ in PSs_{i-1}, PSs_i is formed by all probability-sequence pairs P_i-$<T_1,..,T_{i-1},T_i>$ such that $P_i = \max p(W_i|T_i)p(T_i|T_{i-1})$. The most probable sequence in PSs_n is the one we are looking for.

The processing of a word is taken care of by nesting two calls to *findall/3*.

```
(5)     most_probable_sequence(Words,Sequence) :-
                sequences(Words,[1-[start]],PSs),
                max_key(PSs,P-Sequence1),
                reverse(Sequence1,[start|Sequence]).

        sequences([],PSs,PSs).
        sequences([Word|Words],PSs0,PSs) :-
                findall(PS2,
                                (outprob(Word,T2,PL),
                                findall(P2-[T2,T1|Ts],
                                                (member(P1-[T1|Ts],PSs0),
                                                transprob(T1,T2,PT),
                                                P2 is PL*PT*P1),
                                        PSs),
                                max_key(PSs,PS2)),
                        PSs1),
                sequences(Words,PSs1,PSs).
```

4 Training

In our sample HMM, the probabilities have been invented. There are two well-known methods for *training* the model, i.e. for acquiring the required probabilities automatically. The first method involves collecting statistics from a hand coded training corpus. For example, for each state t and each word form w, the output probabilities can be estimated by taking the number of occurrences of w tagged as t in a training corpus and divide it by the number of occurrences of t. And for each pair of states t_1 and t_2 the transitional probabilities can be estimated by taking the number of occurrences of t_1 being followed by t_2, and divide it by the number of occurrences of t_1. The second method for training a HMM is to estimate these probabilities without a hand coded corpus by means of the *forward-backward* algorithm (see e.g. Charniak 1993).

5 Evaluation

As for the evaluation of methods for statistics-based tagging, researchers are reporting success rates of 90-95% (Charniak 1993). For such a simple approach, this is surprisingly good. Tagging by means of this method is rather efficient as well, provided a good algorithm is used. Learning can be very fast, at least with the first method described above, since it merely involves counting. On the other hand, to work well, both learning methods require a lot of training material.

By adding a tokenizer, procedures for opening and closing files, and procedures for writing to a file, a small but useful standalone automatic part-of-speech tagger system (less than sixty lines of Prolog all together) has been built. Initial testing shows that it is able to tag 10,000 words in 15 seconds (i.e. around 670 words per second) on a SPARCstation 10, running SICStus Prolog 3.0 (compiled code).

6 References

Charniak, E. (1993) *Statistical Language Learning*. Cambridge: MIT Press.

Viterbi, A. J. (1967) Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Programming*. University of Bristol: MIT Press.