Machine Learning for Language Processing
Lecture 2: POS Tagging with HMMs

Stephen Clark

October 6, 2015

The POS Tagging Problem We can't solve the problem by simply compiling a tag dictionary for words, in which each word has a single POS tag. Like most NLP problems, ambiguity is the souce of the difficulty, and must be resolved using the context surrounding each word. Try inputting *I can can a can* into a POS tagger.

Probabilistic Formulation We'll formalise the problem as finding the most probable tag sequence, given the word sequence as input. In early debates regarding statistical methods in AI, the rule-based adherents would always say *sure, but where do the probabilities come from?* We now know the answer to this question: probabilities are estimated from data. However, in the general case, the difficulty of finding appropriate data should not be underestimated. In the POS tagging case, creating manually annotated data, in order to do supervised machine learning, is not too expensive, and is a task a trained linguist can do relatively quickly.

The second question is how to find the arg max, or *what's the search algorithm?* For HMMs there is a very nice dynamic programming solution to the problem of finding the highest scoring tag sequence: the Viterbi algorithm (for sequences).

HMM Tagging Model We'll rewrite the conditional probability of tag sequence given word sequence using Bayes theorem, thereby reversing the direction of the conditional. The reason for doing this is that the likelihood $P(W|T)$, and the prior $P(T)$, are easier to factorise into probabilities that we can estimate than the original conditional, $P(T|W)$. The denominator $P(W)$ can be ignored since W is a constant for the elements of the set we're finding the arg max over.

In order to factorise the two probabilities we first apply the chain rule. Note that the application of the chain rule is *exact*, so no advantage has been gained at this point in terms of ease of estimation: Estimating $P(t_n|t_{n-1}, \ldots, t_1)$ is no easier than estimating the full joint probability $P(t_1, \ldots, t_n)$. However, the application of the chain rule allows us to make independence assumptions.

For the tag sequence probability, we assume that the probability of a tag is only dependent on the previous $n-1$ tags; such a model is called an n-

gram model. For the equation on the slide, the probability of a tag is only dependent on the previous tag, giving a *bigram* model. Note that $P(t_i|t_{i-1})$ is much easier to estimate than the full conditional probability. For the word emission probabilities, we assume that the probability of a word only depends on its corresponding tag.

Clearly both independence assumptions are somewhat drastic. However, relaxing the independence assumptions leads to probabilities which are a) harder to estimate; and b) less efficient to compute with. There is a large literature on exploring this trade-off, for example investigating whether accuracies can be improved by increasing the size of the n-gram for the tag sequence probabilities. Here we find that moving beyond a trigram does not typically help for POS tagging (at least with the sizes of supervised training data available).

N-gram Generative Taggers Note that the HMM is a *generative* model, since we're effectively using the joint distribution: $P(W|T)P(T) = P(W,T)$. Intuitively the model generates both words and tags according to some stochastic process (even though the words are given as input).

Another solution is to use a conditional, or discriminative, model. The advantage of such models is that they are more flexible and do not rely so heavily on independence assumptions. One example of such a model is a maximum entropy tagger, or a tagger based on conditional random fields. We'll encounter such models later in the course. The downside is that the parameters of such models are typically harder to estimate, often requiring numerical methods rather than having a closed-form solution.

Parameter Estimation For an HMM tagger there are two sets of probabilities that need estimating: the tag transition probabilities and the word emission probabilities. Note that the model does not contain paramaters of the form $P(t_i|w_i)$ (because of the application of Bayes theorem), which could be considered counter-intuitive given that the goal is to estimate $P(T|W)$.

There has been lots of work on attempting to build taggers using raw data, treating the problem as an unsupervised machine learning problem, effectively trying to learn clusters of words which correspond to linguistically plausible parts of speech. However, by far the most popular approach to POS tagging is to treat it as a supervised machine learning problem, in which we assume the existence of a manually labelled training set.

Relative Frequency Estimation Suppose I toss a coin 1000 times, and it comes up heads 480 times. An intuitive, and statistically reasonable, estimate of the probability of heads for this coin would be $480/1000 = 0.48$. It turns out that, for the coin tossing case, the relative frequency estimate is also the *maximum likelihood* estimate: 0.48 is the value which makes it most likely that, out of 1000 tosses, heads will appear 480 times. (It is easy to show this formally with some high school calculus.)

The tagging scenario is similar, but with a move from the Bernoulli case, where the value of the relevant random variable (the coin toss) is zero or one, to the Categorical case, where the random variable can take a number of values from a finite set (for the tag transition probabilities it's the tag set; for the word emission probabilities it's the word vocabulary). It turns out that the same mathematics applies to the more general case, and relative frequency estimates for the HMM parameters are maximum likelihood estimates here also. Collins' thesis [2] (Section 2.3) has a very nice presentation of this result.

Smoothing for Tagging There is one aspect of the language estimation problem which is crucially different from the coin tossing scenario. It is easy to toss a coin many times, in order to get a reliable estimate. If a coin has been tossed 1,000 times, we'd be reasonably confident of the relative frequency estimate. However, suppose the coin has only been tossed 5 times, and it came out heads once. How confident would you be now in an estimate of 1/5 for the probability of heads?

What makes statistical modelling difficult in the language case is that we often find ourselves in this *sparse data* position. Since many words are observed rarely, relative frequency estimates are often unreliable. This unreliability is an instance of the machine learning problem of *overfitting* where, intuitively, we're making the probability estimates look *too* much like the data; we're trusting the data too much.

An extreme version of the overfitting problem occurs when events are entirely unseen in the training data, giving frequencies of zero. In the case when the *conditioning* event has not occurred, the relative frequency estimate is undefined, since the denominator is zero ($f(t_{i-1}) = 0$ for estimating $P(t_i|t_{i-1})$). When the event being generated has not occurred, the relative frequency estimate is zero ($f(t_{i-1}, t_i) = 0$ leads to $\hat{P}(t_i|t_{i-1}) = 0$).[1] Zero estimates are problematic because they propagate through the product, giving a zero probability for the whole sequence.

The solution to the overfitting problem is to use *smoothing* techniques, where we take some of the probability mass from the seen events in the training data, and give it to rare and unseen events. Many of the smoothing techniques used in NLP were originally developed for language modelling for speech recognition [1].

The solution on the slide is an example of the *linear interpolation* smoothing technique. The idea is to use progressively more general conditioning events, so that the chances of obtaining zero estimates is reduced. There are various methods available for estimating the λ's. One useful intuition is that we'd like higher values for the relevant λ when the corresponding conditioning event has occurred frequently (the analogue of having tossed the coin many times).

Another related solution is *backing off*. Here, we will use just one of the estimates, rather than mix them, with the decision of which level to use based

[1] $\hat{P}$ is used to denote a (maximum likelihood) estimate of P.

on the frequencies in the relative frequency estimate (e.g. use the first estimate which is not zero).

Better Handling of Unknown Words Suppose we wish to tag a word not seen in training data, e.g. *Trumpington* or *googling*. The capital T in Trumpington — assuming it's not the first word in the sentence, and we're tagging English — is a strong clue the tag should be NNP. Similarly, the *ing* suffix in *googling* is a strong clue the tag should be VBG. One way to adapt the HMM to incorporate such clues is to modify the generative process so that, for unknown words, various features of the word are generated, rather than the word itself. However, making sensible independence assumptions, and deciding which features to generate, is difficult. This difficulty is one of the main motivations for using discriminative models, which allow us to model such features directly.

The Search Problem for Tagging When performing the arg max, the number of tag sequences being searched over grows exponentially with the length of the sentence; simply enumerating all sequences and picking the highest scoring one is not going to work. For a *unigram* model, there is a trivial solution which is efficient and optimal: pick the most probable tag for each word, according to the probability $P(t)P(w|t)$. This algorithm is linear in both the length of the sentence and the size of the tagset.

A Non-Trivial Search Problem Consider now a bigram tagger. Suppose we try and use a similar solution to that for the unigram tagger, picking the most probable tag for each word w_i, according to the probability $P(t_i|t_{i-1})P(w_i|t_i)$ (where t_{i-1} was the tag chosen for the previous word). The example on the slide is designed to show that this algorithm is not optimal. The intuition is that, because of the dependence on the previous tag, the highest scoring tag at any particular point in the left-to-right tagging process could be overtaken by a lower scoring tag when the rest of the sentence is considered. (This intuition does not hold in the unigram case: there is no way that the highest-scoring tag for a word can be overtaken by another tag, because the probability does not depend on the previous tag.)

The Viterbi Algorithm The Viterbi algorithm is a dynamic programming (DP) algorithm, so like all DP algorithms requires the optimal sub-problem property. What this means in practice is that we need to break the larger problem into smaller sub-problems, such that the sub-problems can be solved in the same way (eventually "bottoming out" in a base case which can be solved trivially).

A useful intuition is the following: suppose we want to find the highest-scoring sequence ending at word w_n. Suppose further that we know the highest-scoring (sub-)sequences ending at w_{n-1} for each tag. So we know the highest-scoring sequence ending at w_{n-1} ending in DT; and the highest-scoring sequence ending at w_{n-1} ending in VBG; and the highest-scoring sequence ending at w_{n-1}

ending in NNP; and so on for all tags. From there we need $O(T^2)$ calculations (where T is the size of the tagset), based on the tag transition probabilities between w_{n-1} and w_n, and the word emission probabilities at w_n, to calculate the highest-scoring sequence ending at w_n.

Now apply this idea recursively to find the highest-scoring sequence ending at word w_{n-1} (for each tag). For the base case, calculating the highest-scoring sequence ending at w_1 for each tag is trivial, since there are no previous tags.

Viterbi for a Bigram Tagger The recursion on the slide formalises the intuition given above. In practice, the recursion is implemented by keeping track of the highest-scoring sub-sequences, for each tag and each position, as the tagger moves from left to right (by recording pointers to previous tags); and then the final step is to follow one set of these pointers from the end of the sentence to the start, tracing out the highest-scoring sequence for the whole sentence.

The algorithm is linear in the length of the sentence and quadratic in the size of the tagset. For a trigram tagger, the algorithm is cubic in the size of the tagset (and quartic for a 4-gram tagger, and so on).

Practicalities Taggers based on HMMs can be made highly accurate; e.g. the TnT tagger, which is now over 15 years old, is still competitive, and very fast. In particular, the training can be performed extremely efficiently on large datasets, since the training is essentially just counting. However, the ease of including rich, overlapping features into taggers using discriminative models — the topic of the next lecture — mean that HMM-based taggers are no longer the method of choice.

Readings for Today's Lecture

- Chapters 9 (Markov Models) and 10 (Part-of-Speech Tagging) of Manning and Schütze's Foundations of Statistical NLP; and Chapters 5 (Part-of-Speech Tagging) and 6 (Hidden Markov and Maximum Entropy Models) of Jurafsky and Martin (2nd. Ed.)

References

[1] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. Technical report, 1998.

[2] Michael Collins. *Head-Driven Statistical Models for Natural Language Parsing.* PhD thesis, University of Pennsylvania, 1999.