Machine Learning for Language Processing Lecture 4: The Perceptron for Structured Prediction

Stephen Clark

October 8, 2015

Local and Global Features The local features we have already seen for the maxent tagger can be extended to global features by simply summing up the values of the indicator functions for each position in the sentence. In other words, whilst the local features are indicator functions over local contexts, the global features become *count* functions over the whole sequence.

These global features are used in CRF taggers. The perceptron tagger that we describe here can be thought of as an alternative to the CRF: a global model, but without a probabilistic interpretation and with a different training algorithm. (The decoding algorithm remains the same: the Viterbi algorithm for sequences.)

A Linear Tagging Model The linear tagging model is similar to a MaxEnt model, but without the exponentiation and the normalisation constant. You may be asking: is a linear model powerful enough to solve complex NLP problems? (And didn't Minsky and Papert demonstrate a fundamental weakness of linear perceptrons back in the 60s?) There are two possible responses. The first response is to say that it all depends on the features. With a rich enough feature set, a linear model can always be made powerful enough. Indeed, the whole point of kernels, which we'll encounter later in the lecture, is to allow the use of linear models with extremely complex feature sets.

The second response is to say it's an empirical question. There is a large volume of current work applying non-linear neural networks to various tasks, some of them using many layers in the network; but whether these will turn out to "solve NLP", as some researchers are claiming, remains to be seen. In my experience, the structured perceptron with a large number of features usually provides a very strong baseline for structured prediction tasks [2].

Decoding for a Global Linear Model Decoding is performed in the same way as it was for the HMM and MaxEnt taggers, using the Viterbi algorithm for sequences. The restriction on features, in order that the decoding is efficient, is the same as it was for the MaxEnt tagger: features defined over any part of the input sentence are fine, but features which look over large parts of the generated tag sequence increase the complexity (in the same way as for any n-gram tagger, where the complexity is $O(T^n)$ for a tagset of size T).

The Perceptron Training Algorithm The algorithm is extremely simple: perform a number of passes over the training data, decoding the training instances. If the current model makes a mistake on a training instance, do a simple update. The fact that the weights (potentially) get updated at each instance means that the training is *online*. The fact that no update is performed if the training instance is decoded correctly means that the training is *passive*.

The Training Algorithm (with words) The update works as follows: for each local feature instance in the correct, gold sequence, add 1 its weight value; for each local feature instance in the output given by the tagger, take 1 from its weight value. I use the term *local feature instance* here since the overall update happens globally; i.e. the count of a feature has to be correct across the whole sequence in order for the corresponding feature weight to receive no update.

Averaging Parameters Intuitively, the perceptron training scheme forces the model to perform as well as possible on the training data; but this can lead to overfitting and a lack of generalisation on unseen data. One simple and effective method to combat overfitting is parameter averaging [1]. The set of weights is recorded for each training instance, for each pass over the data, and then averaged. (So for 40,000 training instances and 10 passes, for example, we'd have 400,000 weight vectors to be averaged.) The idea is to mix the earlier weight vectors, which haven't yet converged, with the later ones, which may be overfitted to the training data.

The averaging parameters idea is closely related to the voted perceptron, where, instead of averaging the weights, each instance of the weight vector is used for decoding at test time, and the output is determined through a voting procedure [1]. The obvious disadvantage with this scheme is that, for 40,000 training instances and 10 passes over the training data, the test data has to be decoded separately 400,000 times. The voted perceptron has some theory associated with it, explaining why it is effective.

Theory of the Perceptron The theory of the perceptron [1] shows that, *if the data is separable with some margin*, then the converged model will perform perfectly on the training data. Having (linearly) separable data in this case means that there is some setting of the weights values which will score each

correct output higher than all the incorrect ones (by some margin), for each training instance. What if the training data is not linearly separable? There are guarantees here as well. As long as the data is "close" to being separable, then the number of mistakes on the training data will be small.

Note that, so far, the theoretical discussion has only been in terms of the training data, whereas what we care about is generalising to unseen, test data. There is a pleasing story here as well: if the number of mistakes on the training data is small, then there is a good chance that the number of mistakes on the test data will also be small (making the usual assumptions about the training and test data being drawn from the same distribution, and so on).

A *Ranking* Perceptron The rest of the lecture is geared towards the use of kernels, allowing extremely rich and complex feature sets to be computed with efficiently. We will use the ranking perceptron to motivate the use of kernels, where we assume that a relatively small number of possible outputs have already been generated. The reason to focus on the ranking problem is that we'd like the decoding problem to be trivial, rather than requiring dynamic programming (which would add an extra layer of complexity).

In order to ground the discussion, suppose that we have 1,000 parse trees generated by a statistical parser, for a number of training examples, with the correct tree among those 1,000. The goal of the ranking perceptron is to use models with complex features in order to rank each set of 1,000 trees, so that the correct tree is scored higher than all the incorrect ones (for each training instance). The intention is that the rich models will perform better than the original parsing model (which was likely constrained in terms of the features it could exploit due to the use of DP, or other methods, for navigating the original, very large search space).

Training (with the new notation) The pseudocode on the slide shows just one pass over the training data. It's the same algorithm as before, but using the notation introduced on the previous slide. The purpose of the new notation will become clear.

Note that the calculating the arg max in the ranking scenario is trivial: just exhaustively loop through the candidates $C(s)$ for training sentence s, calculating the score for each.

Perceptron Training (a duel form) The key to the duel form perceptron training is the scoring function $G(\mathbf{x})$, where $\mathbf{x}$ is a feature vector (in our example the features associated with a parse tree). The new scoring function works by taking $\mathbf{x}$ *and comparing it with every other parse tree feature vector in the training data.* $G(\mathbf{x})$ will be high if $\mathbf{x}$ is more similar to the correct parse trees in the data than the incorrect ones.

The training procedure also has a set of weights, but this time a weight α_{ij} for each parse tree vector $\mathbf{x}_{ij}$ in the training data. Note that α_{ij} gets increased if the tree $\mathbf{x}_{ij}$ is returned as the highest scoring one for sentence s_i. The idea is that, if an incorrect parse tree is getting returned as the correct tree for a sentence, then we'd like that tree to have more impact in the training process (in order to fix the incorrect decision).

Equivalence of the two Forms It turns out the two scoring functions $F(\mathbf{x})$ and $G(\mathbf{x})$ are equivalent. (Exercise left for the reader.) So comparing a parse tree feature vector with all the other examples in the training data, and calculating a dot product between the feature vector and (original) weight vector, lead to the same score. It may seem counter-intuitive that there are situations where the first comparison is more efficient than the second, but this is the case if the feature vector is exponentially large. Note that, in the duel form training, *the feature vector does not need to be explicitly represented, as long as the inner product between two parse tree vectors can be calculated.*

Complexity of the two Forms One pass over the data for the standard perceptron learning algorithm takes $O(Td)$ time, where T is the total number of parse trees in the training data (e.g. 1,000 trees for each sentence $\times$ 40,000 sentences), and d is the size of the feature vector (since each dot product calculation between feature and weight vector takes $O(d)$ time).

One pass over the data for the duel form perceptron takes $O(Tnk)$ time, where n is the number of sentences: for each sentence, each of the 1,000 trees needs comparing with all T trees (in order to score each tree). Hence if each tree comparison (dot product between tree vectors) takes $O(k)$ time, then we have $O(n \times 1,000 \times T \times k) = O(Tnk)$.

Hence there can be efficiency gains with the dual form if nk, the number of sentences times the time taken to compute the similarity of objects (e.g. trees), is much smaller than d, the size of the feature vector.

Complexity of Inner Products Can nk ever be much smaller than d? Yes, if the feature vector representation is exponentially large, and if the dot product between such vectors can be calculated efficiently (i.e. without ever explicitly representing the feature vector).

Two examples of feature representations which are exponentially large are representations which a) track all sub-sequences in a sequence (since the number of sub-sequences grows exponentially with the length of the sequence); and b) representations which track all sub-trees in a tree (since the number of sub-trees grows exponentially with the size of the tree).

Tree Kernels A *tree kernel* is a function which returns the number of subtrees in common between two trees. The feature vector representation is one where the basis elements of each vector correspond to a subtree, and the corresponding value is the number of times the subtree appears in the whole tree.

Computation of Subtree Kernel The number of subtrees in common between trees $\mathcal{T}_1$ and $\mathcal{T}_2$ can be calculated efficiently, without ever explicitly representing all subtrees. The algorithm on the slide is a DP algorithm which runs over all pairs of nodes (n_1, n_2) from $\mathcal{T}_1$ and $\mathcal{T}_2$, recursively counting the number of subtrees in common where the subtrees are rooted at n_1 and n_2, $f(n_1, n_2)$.

Two cases are straightforward. If the context-free productions with parents n_1 and n_2 are different, then the number of subtrees in common rooted at n_1 and n_2 has to be zero. For the base case, i.e. when n_1 and n_2 are pre-terminals, $f(n_1, n_2) = 1$ if the productions rooted at n_1 and n_2 are the same.

For the recursive case, if the productions rooted at n_1 and n_2 are the same, then we get a count for the production itself, which combines with a recursive call on each of the child pairs from n_1 and n_2. (Exercise for the reader: run through some examples to convince yourself that the recursive case is correct.)

Readings for Today's Lecture

- Michael Collins and Nigel Duffy. New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron. ACL 2002.

References

[1] Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of EMNLP*, pages 1–8, Philadelphia, USA, 2002.

[2] Yue Zhang and Stephen Clark. Syntactic processing using the generalized perceptron and beam search. *Computational Linguistics*, 37(1):105–151, 2011.