

Sequence labeling and beyond (cont'd)

Structured Perceptron

Remember the CRF... We had the following basic algorithm for optimization:

1. Increase weights around correct paths
2. Decrease weights around incorrect paths
(decrease proportional to the probability of hitting those paths)

This is a general theme for structured prediction models. Other algorithms (that we don't discuss) such as max-margin Markov networks, structured SVMs, etc., all embody this basic premise, but with slightly different parentheticals on step (2).

The problem with CRFs is that for most graph structures, step 2 above is intractable. So we're going to replace it with something more tractable. This leads to the *structured perceptron*.

1. Given data $(x_1, y_1), \dots, (x_N, y_N)$ with structured y s.
2. Initialize a weight vector $w \leftarrow 0$
3. For a number of iterations:
 - (a) For each $1 \leq n \leq N$:
 - i. Compute predicted output $\hat{y} = \operatorname{argmax}_{y \in \mathcal{Y}} w^\top \Phi(x_n, y)$
 - ii. If $\hat{y} \neq y_n$, perform update:

$$w \leftarrow w + \Phi(x_n, y_n) - \Phi(x_n, \hat{y})$$
 - iii. (Optional: average weights)

This is just like the perceptron for binary classification, but extended to structured prediction. Note that for sequence labeling problems, we write $\Phi(x_n, y_n)$ for the sum of all features over all substructures: $\Phi(x_n, y_n) = \sum_m \Phi(x_n, y_{n,m}, y_{n,m-1})$ for instance.

Note that in comparison to the CRF, we have reduced the requirement of summing over all possible paths (forward-backward) to the requirement of computing the best path (Viterbi). One way of thinking about this is to consider the gradient of the log-likelihood for the CRF:

$$\nabla_w \ell(y_n | x_n) = \sum_m \left\{ \Phi(x_n, y_{n,m}, y_{n,m-1}) - \sum_{l,p} p(y_m = l, y_{m-1} = p) \Phi(x_n, l, p) \right\}$$

We obtain something that identical to the perceptron update if we assume that a single path—the best path—has all of the probability mass. This is exactly the same relationship we had between logistic regression and the (classification) perceptron from the first part of class.

One question that arises is: what if the argmax isn't even tractable? The first solution is to use a best-guess y , found using some search process, instead of the argmax. But, in general, we can do better...

Search-based Structured Prediction

A problem near to my heart in machine translation: given an Arabic sentence a , find the best possible English translation e . I.e., for some scoring function f , for an input a , find $\hat{e} = \operatorname{argmax}_e f(e, a)$. In general, this will require considering a gigantic number of possible e : even for a dumbed down model, the problem is NP-hard. This is the case for many problems in many domains (eg. biology). An additional problem is that the loss functions we care about are complex.

So, what do we do in practice when we want to solve an NP-hard problem? We use an approximate search technique (or approximation algorithm, but they're usually pretty similar).

Search(a, f):

1. Initialize a max-queue $Q = \{\epsilon\}$
2. While Q isn't empty:
 - (a) Remove h , the top scoring partial hypothesis from Q
 - (b) If h represents a complete hypothesis, return it
 - (c) For all possible ways of extending $h \mapsto h'$,
 - i. Compute a score $s' = f(a, h')$
 - ii. Add h' to Q with score s'
 - (d) If Q is too large, drop some low-scoring elements from it.

In the translation example, we may consider a left-to-right search over e . In each "extension" step, we add a single word to e . The score will then depend on: (a) how grammatical is e up to the current position and (b) how well does it reflect a translation of a substring of a ?

How can we do learning in such a system? Well, we can compute scores using a learned model. I.e., use $s' = f(a, h')$, where f is a learned function. The remaining question is: how can we learn f based on training data of the form of pairs of inputs and outputs?

What do we want? We want a function f so that $\ell(e, \text{Search}(a, f))$ is as low as possible, where ℓ is a loss function. We're going to do this via reductions. Specifically, we will reduce to cost-sensitive (multiclass) classification (from which we can apply, eg., weighted all pairs and costing).

To simplify the problem, consider *greedy search*:

GreedySearch(a, f):

1. Initialize output $e = \epsilon$
2. While e is not complete:
 - (a) Choose an *action* we can take from e , $f(\Phi(a, e))$.
 - (b) Update $e \leftarrow \langle e, a \rangle$
3. Return e

Here, f looks just like a classifier over (features of) *actions*.

So the question is how can we create training data to train such an f ?

Obvious attempt number one:

1. Initialize output $e = \epsilon$
2. While e is not complete
 - (a) Make a training example based on $\Phi(a, e)$ where the optimal action is the positive class and all other actions are the negative class
 - (b) Update $e \leftarrow \langle e, \text{optimal action} \rangle$
3. Train classifier on all classification examples

There are (at least) two problems with this:

1. How do we incorporate loss information—some mistakes won't be as bad as others?
2. This doesn't solve the label-bias problem: it makes it worse! We're training assuming we make all previous decisions optimally.

We'll solve both of these problems at once, but will focus on the latter. The problem is chicken-and-egg. We want to learn a classifier (action chooser) that does well when previous decisions are made by that classifier. As in most chicken-and-egg problems, we solve it by iteration. Idea:

1. Train a classifier $f^{(1)}$ assuming all decisions are made optimally
2. Train a classifier $f^{(2)}$ assuming most decisions are made optimally, but some are made by $f^{(1)}$
3. Train a classifier $f^{(2)}$ assuming many decisions are made optimally, but some are made by $f^{(1)}$ and some by $f^{(2)}$
4. ...
5. Train a classifier $f^{(T)}$ assuming no decisions are made optimally, but some are made by $f^{(1)}, \dots, f^{(T-1)}$

To incorporate the loss, we will predict out to the end, and then compute the loss for the whole output.

This yields the following algorithm, called SEARN:

1. Set π to be an "optimal classifier" that always makes the correct decisions (possible on training data)
2. For T iterations:
 - (a) For each (x_n, y_n) in the training data:
 - i. Initialize $y = \epsilon$
 - ii. While y is not complete:
 - A. For each action a ,
 - Compute $\hat{y}_a = \text{result of running } \pi \text{ to the end}$
 - Compute cost $\ell(y_n, \hat{y}_a)$
 - B. Make a training example for all possible actions with costs given as above
 - C. Update $e \leftarrow \langle e, \pi(x_n, y) \rangle$
 - (b) Train classifier $f^{(T)}$ on all classification examples
 - (c) Set $\pi = (1 - \beta)\pi + \beta f^{(T)}$
3. Return π , removing dependence on training data

The theoretical result (not proven here) is that if β is sufficiently small, then this will converge to something not-too-far from optimal (assuming the underlying classifiers do a good job).