Day 3

Learning Structured Predictors

In this class, we will continue to focus on sequence classification, but instead of following a *generative* approach (like in the previous chapter) we move towards *discriminative* approaches.

Table 3.1 shows how the models for classification have counterparts in *sequential* classification. In fact, in the last chapter we discussed the Hidden Markov model, which can be seen as a generalization of the Naïve Bayes model for sequences. In this chapter, we will see a generalization of the Perceptron algorithm for sequence problems (yielding the Structured Perceptron, Collins 2002) and a generalization of Maximum Entropy model for sequences (yielding Conditional Random Fields, Lafferty et al. 2001). Note that both these generalizations are not specific for sequences and can be applied to a wide range of models (we will see in tomorrow's lecture how these methods can be applied to parsing). Although we will not cover all the methods described in Chapter 1, bear in mind that all of those have a structured counterpart. It should be intuitive after this lecture how those methods could be extended to structured problems, given the perceptron example. Before we explain the particular methods, the next section will talk a bit about feature representation for

Classification	**Sequences**
Generative	
Naïve Bayes 1.2	Hidden Markov Models 2.1
Discriminative	
Perceptron 1.4.1	Structured Perceptron 3.2
Maximum Entropy 1.5.1	Conditional Random Fields 3.3

Table 3.1: Summary of the methods that we will be covering this lecture.

Condition	Name
$\mathbf{y}_i = l$ & $t = 0$	Initial State
$\mathbf{y}_i = l$ & $\mathbf{y}_{i-1} = m$	Transition Features
$\mathbf{y}_i = l$ & $\mathbf{y}_{i-1} = m$ & $t = N$	Final Transition Features
$\bar{\mathbf{x}}_i = a$ & $\mathbf{y}_i = l$	Observation Features

Table 3.2: IDFeatures feature set. This set replicates the features used by the HMM model.

sequences.

$$\arg\max_{\bar{\mathbf{y}}} = p_\theta(\bar{\mathbf{y}}|\bar{\mathbf{x}}) = \theta \cdot f(\bar{\mathbf{x}}, \bar{\mathbf{y}}) \tag{3.1}$$

As in the previous section, $\bar{\mathbf{y}}$ is a sequence so the maximization is over an exponential number of objects, making it intractable. Again we will assume a first order markov independence assumption, and so the features will decompose as the model. So Equation 3.1 can be written as:

$$\arg\max_{\bar{\mathbf{y}}} = \sum_N \sum_{\bar{\mathbf{y}}} \theta \cdot f(n, \mathbf{y}_n, \bar{\mathbf{x}}_n) + \sum_N \sum_{\mathbf{y}_n \in \mathbf{Y}} \theta \cdot f(n, \mathbf{y}_n, \mathbf{y}_{n-1}, \bar{\mathbf{x}}_n) \tag{3.2}$$

3.1 Feature Extraction

In this section we will define two simple feature sets. The first one will only use identity features, and will mimic the features used by the HMM model from the previous section. This will allow to directly compare the performance of a generative vs a discriminative approach. Note that although not required, all the features we will use in this section are binary features (0-1), indicating the presence or absence of a given condition.

Example 3.1 *Simple ID Feature set containing the same features as an HMM model.*

```
0 Ms./NOUN NF: id:Ms.::NOUN init_tag:NOUN
1 Haag/NOUN NF: id:Haag::NOUN EF: prev_tag:NOUN::NOUN
2 plays/VERB NF: id:plays::VERB EF: prev_tag:NOUN::VERB
3 Elianti/NOUN NF: id:Elianti::NOUN EF: prev_tag:VERB::NOUN
4 ./. NF: id:.::. EF: last_prev_tag:NOUN::.
```

Table 3.2 depicts the features that are implicit in the HMM, which was the subject of the previous chapter. These features are indicators of initial, final, observation and transition events. The fact that we were using a generative

Condition	Name
$\mathbf{y}_i = l$ & $t = 0$	Initial State
$\mathbf{y}_i = l$ & $\mathbf{y}_{i-1} = m$	Transition Features
$\mathbf{y}_i = l$ & $\mathbf{y}_{i-1} = m$ & $t = N$	Final Transition Features
$\bar{\mathbf{x}}_i = a$ & $\mathbf{y}_i = l$	Observation Features
$\bar{\mathbf{x}}_i = a$ & a is uppercased & $\mathbf{y}_i = l$	Uppercase Features
$\bar{\mathbf{x}}_i = a$ & a contains digit & $\mathbf{y}_i = l$	Digit Features
$\bar{\mathbf{x}}_i = a$ & a contains hyphen & $\mathbf{y}_i = l$	Hyphen Features
$\bar{\mathbf{x}}_i = a$ & $a_{[0..i]} \forall i \in [1,2,3]$ $\mathbf{y}_i = l$	Prefix Features
$\bar{\mathbf{x}}_i = a$ & $a_{[N-i..N]} \forall i \in [1,2,3]$ & $\mathbf{y}_i = l$	Suffix Features

Table 3.3: Extended feature set. Some features in this set could not be included in the HMM model.

model has forced us (in some sense) to make strong independence assumptions. However, since we now move to a discriminative approach, where we model $P(\bar{\mathbf{y}}|\bar{\mathbf{x}})$ rather than $P(\bar{\mathbf{x}}, \bar{\mathbf{y}})$, we are not tied anymore to some of these assumptions. In particular:

- We may use "overlapping" features, *e.g.*, features that fire simultaneously for many instances. For example, we can use a feature for a word and another for prefixes and suffixes of that word. This would lead to an awkward model if we wanted to insist on a generative approach.

- We may use features that depend arbitrarily on the entire *input* sequence $\bar{\mathbf{x}}$. On the other hand, we still need to resort to "local" features with respect to the *outputs* (*e.g.* looking only at consecutive state pairs), otherwise decoding algorithms will become more expensive.

Table 3.3 shows examples of features that are traditionally used in POS tagging with discriminative models. Of course, we could have much more complex features, looking arbitrarily to the input sequence. We are not going to have them in this exercise only for performance reasons (to have less features and smaller caches).

Example 3.2

```
0 Ms./NOUN NF: id:Ms.::NOUN uppercased::NOUN suffix:.::NOUN
   suffix:s.::NOUN prefix:M::NOUN prefix:Ms::NOUN init_tag:NOUN
1 Haag/NOUN NF: id:Haag::NOUN uppercased::NOUN suffix:g::NOUN
   suffix:ag::NOUN suffix:aag::NOUN prefix:H::NOUN prefix:Ha::
   NOUN prefix:Haa::NOUN rare::NOUN EF: prev_tag:NOUN::NOUN
2 plays/VERB NF: id:plays::VERB EF: prev_tag:NOUN::VERB
3 Elianti/NOUN NF: id:Elianti::NOUN uppercased::NOUN suffix:i::
   NOUN suffix:ti::NOUN suffix:nti::NOUN prefix:E::NOUN prefix:
   El::NOUN prefix:Eli::NOUN rare::NOUN EF: prev_tag:VERB::NOUN
4 ./. NF: id:.::. EF: last_prev_tag:NOUN::.
```

We consider two kinds of features: *node features*, which form a vector $\boldsymbol{f}_N(\bar{\mathbf{x}}, y_i)$, and *edge features*, which form a vector $\boldsymbol{f}_E(\bar{\mathbf{x}}, y_i, y_{i-1})$.[1] These feature vectors will receive parameter vectors $\boldsymbol{\theta}_N$ and $\boldsymbol{\theta}_E$. Similarly as in the previous chapter, we consider:

- *Node Potentials.* These are scores for a state at a particular position. They are given by
$$\psi_V(\bar{\mathbf{x}}, \mathbf{y}_i) = \exp(\boldsymbol{\theta}_V \cdot \boldsymbol{f}_V(\bar{\mathbf{x}}, \mathbf{y}_i)). \tag{3.3}$$

- *Edge Potentials.* These are scores for the transitions. They are given by
$$\psi_E(\bar{\mathbf{x}}, \mathbf{y}_i, \mathbf{y}_{i-1}) = \exp(\boldsymbol{\theta}_E \cdot \boldsymbol{f}_E(\bar{\mathbf{x}}, \mathbf{y}_i, \mathbf{y}_{i-1})). \tag{3.4}$$

Let $\boldsymbol{\theta} = (\boldsymbol{\theta}_N, \boldsymbol{\theta}_E)$. The conditional probability $P(\bar{\mathbf{y}}|\bar{\mathbf{x}})$ is then defined as follows:

$$\begin{aligned} P(\bar{\mathbf{y}}|\bar{\mathbf{x}}) &= \frac{1}{Z(\boldsymbol{\theta}, \bar{\mathbf{x}})} \exp\left(\sum_i \boldsymbol{\theta}_V \cdot \boldsymbol{f}_V(\bar{\mathbf{x}}_i, \mathbf{y}_i) + \sum_i \boldsymbol{\theta}_E \cdot \boldsymbol{f}_E(\bar{\mathbf{x}}_i, \mathbf{y}_i, \mathbf{y}_{i-1})\right) && (3.5) \\ &= \frac{1}{Z(\boldsymbol{\theta}, x)} \prod_i \psi_V(\bar{\mathbf{x}}_i, \mathbf{y}_i)\psi_E(\bar{\mathbf{x}}_i, \mathbf{y}_i, \mathbf{y}_{i-1}), && (3.6) \end{aligned}$$

where

$$Z(\boldsymbol{\theta}, x) = \sum_{\mathbf{y} \in \mathbf{Y}} \prod_i \psi_V(\bar{\mathbf{x}}_i, \mathbf{y}_i)\psi_E(\bar{\mathbf{x}}_i, \mathbf{y}_i, \mathbf{y}_{i-1}) \tag{3.7}$$

is the partition function.

There are three important problems that need to be solved:

1. Given $\bar{\mathbf{x}}$, computing the most likely output sequence $\bar{\mathbf{y}}$ (the one which maximizes $P(\bar{\mathbf{y}}|\bar{\mathbf{x}})$.

2. Compute the posterior marginals $P(\mathbf{y}_i|\bar{\mathbf{x}})$ at each position i.

3. Compute the partition function.

Interestingly, all these problems can be solved by using the same algorithms (just changing the potentials) that were already implemented for HMMs: the Viterbi algorithm (for 1) and the forward-backward algorithm (for 2–3).

[1] To make things simpler, we will assume later on that edge features do not depend on the input $\bar{\mathbf{x}}$—but they could, without changing at all the decoding algorithm.

Algorithm 10 Averaged Structured perceptron

input: dataset $\mathcal{D}$, number of rounds T
initialize $w^1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  choose $m = m(t)$ randomly
  take training pair $(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m)$ and predict using the current model:

$$\hat{\bar{\mathbf{y}}} \leftarrow \arg\max_{\bar{\mathbf{y}}'} w^t \cdot f(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}')$$

  update the model: $w^{t+1} \leftarrow w^t + f(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m) - f(\bar{\mathbf{x}}^m, \hat{\bar{\mathbf{y}}})$
end for
output: the averaged model $\hat{w} \leftarrow \frac{1}{T}\sum_{t=1}^{T} w^t$

3.2 Structured Perceptron

The structured perceptron (Collins, 2002), namely its averaged version is a very simple algorithm that relies on Viterbi decoding and very simple additive updates. In practice this algorithm is very easy to implement and behaves remarkably well in a variety of problems. These two characteristics make the structured perceptron algorithm a natural first choice to try and test a new problem or a new feature set.

Recall what you learned from §1.4.1 on the perceptron algorithm and compare it against the structured perceptron (Algorithm 10).

There are only two differences:

- Instead of finding $\arg\max_{y' \in \mathcal{Y}}$ for a given variable, it finds the $\arg\max_{\bar{\mathbf{y}}'}$, the best sequence. We can do this by using the Viterbi algorithm with the node and edge potentials (actually, the log of those potentials) defined in Eqs. 3.3–3.4, along with the assumption that the features decompose as the model, as explained in the previous section.

- Instead of updating the features for the entire y' (in this case $\bar{\mathbf{y}}$) we update the features only at the positions were the labels are different.

Exercise 3.1 *In this exercise you will test the structured perceptron algorithm using different feature sets for Part-of-Speech Tagging.*

Start by loading the corpus and creating an IDFeature class. Next initialize the perceptron and train the algorithm.

```
import sys
sys.path.append("readers/" )
sys.path.append("sequences/" )

import pos_corpus as pcc
```

```
import id_feature as idfc

import structured_perceptron as spc


posc = pcc.PostagCorpus("en",max_sent_len=15,train_sents=1000,
    dev_sents=200,test_sents=200)
id_f = idfc.IDFeatures(posc)
id_f.build_features()
sp = spc.StructuredPercetron(posc,id_f)
sp.nr_rounds = 20
sp.train_supervised(posc.train.seq_list)

Epoch: 0 Accuracy: 0.617797
Epoch: 1 Accuracy: 0.797775
Epoch: 2 Accuracy: 0.864115
Epoch: 3 Accuracy: 0.901794
Epoch: 4 Accuracy: 0.925644
Epoch: 5 Accuracy: 0.932659
Epoch: 6 Accuracy: 0.938872
Epoch: 7 Accuracy: 0.946087
Epoch: 8 Accuracy: 0.949193
Epoch: 9 Accuracy: 0.950696
Epoch: 10 Accuracy: 0.952701
Epoch: 11 Accuracy: 0.952600
Epoch: 12 Accuracy: 0.956910
Epoch: 13 Accuracy: 0.956108
Epoch: 14 Accuracy: 0.956408
Epoch: 15 Accuracy: 0.958413
Epoch: 16 Accuracy: 0.957110
Epoch: 17 Accuracy: 0.959014
Epoch: 18 Accuracy: 0.959315
Epoch: 19 Accuracy: 0.960216
```

Now evaluate the learned model on both the development and test set.

```
pred_train = sp.viterbi_decode_corpus(posc.train.seq_list)
pred_dev = sp.viterbi_decode_corpus(posc.dev.seq_list)
pred_test = sp.viterbi_decode_corpus(posc.test.seq_list)
eval_train = sp.evaluate_corpus(posc.train.seq_list,pred_train)
eval_dev = sp.evaluate_corpus(posc.dev.seq_list,pred_dev)
eval_test = sp.evaluate_corpus(posc.test.seq_list,pred_test)
print "Structured Percetron - ID Features Accuracy Train: %.3f
    Dev: %.3f Test: %.3f"%(eval_train,eval_dev,eval_test)

Out[]: Structured Percetron - ID Features Accuracy Train: 0.867
     Dev: 0.831 Test: 0.790
```

Compare with the results achieved with the HMM model.

```
Best Smoothing 1.000000 --  Test Set Accuracy: Posterior Decode
     0.809, Viterbi Decode: 0.777
```

Even using a similar feature set the perceptron yields better results. Perform some error analysis and figure out what are the main errors the perceptron is making. Compare them with the errors made by the HMM model. (Hint: use the methods developed in the previous lecture to help you with the error analysis).

Exercise 3.2 *Repeat the previous exercise using the extended feature set. Compare the results.*

```
import extended_feature as exfc

ex_f = exfc.ExtendedFeatures(posc)
ex_f.build_features()
sp = spc.StructuredPercetron(posc,ex_f)
sp.nr_rounds = 20
sp.train_supervised(posc.train.seq_list)

Epoch: 0 Accuracy: 0.638741
Epoch: 1 Accuracy: 0.807596
Epoch: 2 Accuracy: 0.876541
Epoch: 3 Accuracy: 0.907406
Epoch: 4 Accuracy: 0.921836
Epoch: 5 Accuracy: 0.939974
Epoch: 6 Accuracy: 0.940575
Epoch: 7 Accuracy: 0.948893
Epoch: 8 Accuracy: 0.948893
Epoch: 9 Accuracy: 0.950095
Epoch: 10 Accuracy: 0.954404
Epoch: 11 Accuracy: 0.957110
Epoch: 12 Accuracy: 0.954605
Epoch: 13 Accuracy: 0.956910
Epoch: 14 Accuracy: 0.956509
Epoch: 15 Accuracy: 0.956609
Epoch: 16 Accuracy: 0.958012
Epoch: 17 Accuracy: 0.959014
Epoch: 18 Accuracy: 0.957411
Epoch: 19 Accuracy: 0.958413

pred_train = sp.viterbi_decode_corpus(posc.train.seq_list)
pred_dev = sp.viterbi_decode_corpus(posc.dev.seq_list)
pred_test = sp.viterbi_decode_corpus(posc.test.seq_list)

```

```
eval_train = sp.evaluate_corpus(posc.train.seq_list,pred_train)
eval_dev = sp.evaluate_corpus(posc.dev.seq_list,pred_dev)
eval_test = sp.evaluate_corpus(posc.test.seq_list,pred_test)

print "Structured Percetron - Extended Features Accuracy Train:
     %.3f  Dev: %.3f Test:   %.3f"%(eval_train,eval_dev,
    eval_test)

Structured Percetron - Extended Features Accuracy Train: 0.946
     Dev: 0.868 Test:   0.840
```

Compare the errors obtained with the two different feature sets. Perform some feature analysis, what errors were correct by using more features? Can you think of other features to use to solve the errors found?

3.3 Conditional Random Fields

Conditional Random Fields (CRF) (Lafferty et al., 2001) can be seen as an extension of the Maximum Entropy (ME) models to structured problems.[2]

CRFs are *globally* normalized models: the probability of a given sentence is given by Equation 3.5. Going from a maximum entropy model (in multi-class classification) to a CRF mimics the transition discussed above from perceptron to structured perceptron:

- Instead of finding the posterior marginal $P(y'|x)$ for a given variable, it finds the posterior marginals for all factors (nodes and edges), $P(\bar{\mathbf{y}}_i|\bar{\mathbf{x}})$ and $P(\bar{\mathbf{y}}_i, \bar{\mathbf{y}}_{i-1}|\bar{\mathbf{x}})$. We can compute this quantities by using the forward-backward algorithm with the node and edge potentials defined in Eqs. 3.3–3.4, along with the assumption that the features decompose as the model, as explained in the previous section.

- The features are updated factor wise (i.e., for each node and edge).

Algorithm 11 shows the pseudo code to optimize a CRF with a batch gradient method (in the exercise, we will use a quasi-Newton method, L-BFGS). Again, we can also take an online approach to optimization, but here we will stick with the batch one.

Exercise 3.3 *Repeat Exercises 3.1–3.2 using a CRF model instead of the perceptron algorithm. Report the results.*

Here is the code for the simple feature set:

[2] An earlier, less successful, attempt to perform such an extension was via Maximum Entropy Markov models (MEMM) (McCallum et al., 2000). There each factor (a node or edge) is a *locally* normalized maximum entropy model. A shortcoming of MEMMs is its so-called *labeling bias* (Bottou, 1991), which makes them biased towards states with few successor states (see Lafferty et al. (2001) for more information).

Algorithm 11 Batch Gradient Descent for Conditional Random Fields

input: $\mathcal{D}$, λ, number of rounds T, learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{\theta}^1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  for $m = 1$ **to** M **do**
    take training pair (x^m, y^m) and compute conditional probabilities using the current model, for each $\bar{\mathbf{y}}$:

$$P_{\boldsymbol{\theta}^t}(\bar{\mathbf{y}}|\bar{\mathbf{x}}) = \frac{1}{Z(\boldsymbol{\theta}^t, \bar{\mathbf{x}})} \exp\left(\sum_i \boldsymbol{\theta}_V^t \cdot \boldsymbol{f}_V(\bar{\mathbf{x}}, \mathbf{y}_i) + \sum_i \boldsymbol{\theta}_E^t \cdot \boldsymbol{f}_E(\bar{\mathbf{x}}, \mathbf{y}_i, \mathbf{y}_{i-1})\right)$$

    compute the feature vector expectation:

$$E_{\boldsymbol{\theta}^t}[\boldsymbol{f}(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m)] = \sum_{\bar{\mathbf{y}}} P_{\boldsymbol{\theta}^t}(\bar{\mathbf{y}}^m|\bar{\mathbf{x}}^m)\boldsymbol{f}(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m)$$

  end for
  choose the stepsize η_t using, *e.g.*, Armijo's rule
  update the model:

$$\boldsymbol{\theta}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{\theta}^t + \eta_t M^{-1} \sum_{m=1}^{M} \left(\boldsymbol{f}(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m) - E_{\boldsymbol{\theta}^t}[\boldsymbol{f}(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m)]\right)$$

end for
output: $\hat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{T+1}$

```python
import crf_batch as crfc
posc = pcc.PostagCorpus("en",max_sent_len=15,train_sents=1000,
    dev_sents=200,test_sents=200)
id_f = idfc.IDFeatures(posc)
id_f.build_features()


crf = crfc.CRF_batch(posc,id_f)
crf.train_supervised(posc.train.seq_list)

pred_train = crf.viterbi_decode_corpus(posc.train.seq_list)
pred_dev = crf.viterbi_decode_corpus(posc.dev.seq_list)
pred_test = crf.viterbi_decode_corpus(posc.test.seq_list)

eval_train = crf.evaluate_corpus(posc.train.seq_list,pred_train
    )
eval_dev = crf.evaluate_corpus(posc.dev.seq_list,pred_dev)
eval_test = crf.evaluate_corpus(posc.test.seq_list,pred_test)

```

```
print "CRF - ID Features Accuracy Train: %.3f Dev: %.3f Test:
    %.3f"%(eval_train,eval_dev,eval_test)
Out[]: CRF - ID Features Accuracy Train: 0.920 Dev: 0.863 Test:
     0.830
```

Here is the code for the extended feature set:

```

posc = pcc.PostagCorpus("en",max_sent_len=15,train_sents=1000,
    dev_sents=200,test_sents=200)
ex_f = exfc.ExtendedFeatures(posc)
ex_f.build_features()


crf = crfc.CRF_batch(posc,ex_f)
crf.train_supervised(posc.train.seq_list)

pred_train = crf.viterbi_decode_corpus(posc.train.seq_list)
pred_dev = crf.viterbi_decode_corpus(posc.dev.seq_list)
pred_test = crf.viterbi_decode_corpus(posc.test.seq_list)

eval_train = crf.evaluate_corpus(posc.train.seq_list,pred_train
    )
eval_dev = crf.evaluate_corpus(posc.dev.seq_list,pred_dev)
eval_test = crf.evaluate_corpus(posc.test.seq_list,pred_test)

print "CRF - Extended Features Accuracy Train: %.3f Dev: %.3f
    Test: %.3f"%(eval_train,eval_dev,eval_test)

Out[]: CRF - Extended Features Accuracy Train: 0.924 Dev: 0.872
     Test: 0.831
```