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

<table>
<thead>
<tr>
<th>Classification</th>
<th>Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generative</td>
<td></td>
</tr>
<tr>
<td>Naïve Bayes [1.2]</td>
<td>Hidden Markov Models [2.1]</td>
</tr>
<tr>
<td>Discriminative</td>
<td></td>
</tr>
<tr>
<td>Perceptron [1.4.1]</td>
<td>Structured Perceptron [3.2]</td>
</tr>
<tr>
<td>Maximum Entropy [1.5.1]</td>
<td>Conditional Random Fields [3.3]</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of the methods that we will be covering this lecture.
\[
\arg\max_{\bar{y}} = p_{\theta}(\bar{y}|\bar{x}) = \theta \cdot f(\bar{x}, \bar{y}) \tag{3.1}
\]

As in the previous section, $\bar{y}$ 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:

\[
\arg\max_{\bar{y}} = \sum_{N} \sum_{\bar{y}} \theta \cdot f(n, y_n, \bar{x}_n) + \sum_{N} \sum_{y_n \in Y} \theta \cdot f(n, y_n, y_{n-1}, \bar{x}_n) \tag{3.2}
\]

### 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.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_i = l$ &amp; $t = 0$</td>
<td>Initial State</td>
</tr>
<tr>
<td>$y_i = l$ &amp; $y_{i-1} = m$</td>
<td>Transition Features</td>
</tr>
<tr>
<td>$y_i = l$ &amp; $y_{i-1} = m$ &amp; $t = N$</td>
<td>Final Transition Features</td>
</tr>
<tr>
<td>$\bar{x}_i = a$ &amp; $y_i = l$</td>
<td>Observation Features</td>
</tr>
</tbody>
</table>

Table 3.2: IDFeatures feature set. This set replicates the features used by 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 \( P(\vec{y}|\vec{x}) \) rather than \( P(\vec{x}, \vec{y}) \), 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 \( \vec{x} \). 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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_i = l ) &amp; ( t = 0 )</td>
<td>Initial State</td>
</tr>
<tr>
<td>( y_i = l ) &amp; ( y_{i-1} = m )</td>
<td>Transition Features</td>
</tr>
<tr>
<td>( y_i = l ) &amp; ( y_{i-1} = m ) &amp; ( t = N )</td>
<td>Final Transition Features</td>
</tr>
<tr>
<td>( \vec{x}_i = a ) &amp; ( y_i = l )</td>
<td>Observation Features</td>
</tr>
<tr>
<td>( \vec{x}_i = a ) &amp; ( a ) is uppercased &amp; ( y_i = l )</td>
<td>Uppercase Features</td>
</tr>
<tr>
<td>( \vec{x}_i = a ) &amp; ( a ) contains digit &amp; ( y_i = l )</td>
<td>Digit Features</td>
</tr>
<tr>
<td>( \vec{x}_i = a ) &amp; ( a ) contains hyphen &amp; ( y_i = l )</td>
<td>Hyphen Features</td>
</tr>
<tr>
<td>( \vec{x}_i = a ) &amp; ( a ) (</td>
<td>N−i..N</td>
</tr>
<tr>
<td>( \vec{x}_i = a ) &amp; ( a ) (</td>
<td>N−i..N</td>
</tr>
</tbody>
</table>

Table 3.3: Extended feature set. Some features in this set could not be included in the HMM model.
We consider two kinds of features: node features, which form a vector $f_N(\bar{x}, y_i)$, and edge features, which form a vector $f_E(\bar{x}, y_i, y_{i-1})$. These feature vectors will receive parameter vectors $\theta_N$ and $\theta_E$. Similarly as in the previous chapter, we consider:

- **Node Potentials.** These are scores for a state at a particular position. They are given by
  \[ \psi_V(\bar{x}, y_i) = \exp(\theta_V \cdot f_V(\bar{x}, y_i)). \] (3.3)

- **Edge Potentials.** These are scores for the transitions. They are given by
  \[ \psi_E(\bar{x}, y_i, y_{i-1}) = \exp(\theta_E \cdot f_E(\bar{x}, y_i, y_{i-1})). \] (3.4)

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

\[
P(\bar{y}|\bar{x}) = \frac{1}{Z(\theta, \bar{x})} \exp \left( \sum_i \theta_V \cdot f_V(\bar{x}_i, y_i) + \sum_i \theta_E \cdot f_E(\bar{x}_i, y_i, y_{i-1}) \right) \tag{3.5}
\]
\[
= \frac{1}{Z(\theta, x)} \prod_i \psi_V(\bar{x}_i, y_i) \psi_E(\bar{x}_i, y_i, y_{i-1}), \tag{3.6}
\]

where

\[
Z(\theta, x) = \sum_{y \in Y} \prod_i \psi_V(\bar{x}_i, y_i) \psi_E(\bar{x}_i, y_i, y_{i-1}) \tag{3.7}
\]
is the partition function.

There are three important problems that need to be solved:

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

2. Compute the posterior marginals $P(y_i|\bar{x})$ 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).

---

To make things simpler, we will assume later on that edge features do not depend on the input $\bar{x}$—but they could, without changing at all the decoding algorithm.
Algorithm 10 Averaged Structured perceptron

1: input: dataset \( D \), number of rounds \( T \)
2: initialize \( w^1 = 0 \)
3: for \( t = 1 \) to \( T \) do
4: choose \( m = m(t) \) randomly
5: take training pair \((\bar{x}^m, \bar{y}^m)\) and predict using the current model:
   \[
   \hat{y} \leftarrow \arg \max_{\bar{y}'} \bar{w}^t \cdot f(\bar{x}^m, \bar{y}')
   \]
6: update the model:
   \[
   w^{t+1} \leftarrow w^t + f(\bar{x}^m, \bar{y}^m) - f(\hat{x}^m, \hat{y})
   \]
7: end for
8: output: the averaged model \( \bar{w} \leftarrow \frac{1}{T} \sum_{t=1}^{T} w^t \)

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 \( \arg \max_{y' \in Y} \) for a given variable, it finds the \( \arg \max_{\bar{y}'} \), 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 \( \bar{y} \)) 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.

```python
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.StructuredPerceptron(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 Perceptron - ID Features Accuracy Train: %.3f Test: %.3f"
    % (eval_train, eval_dev, eval_test)

Out[]: Structured Perceptron - 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.

```python
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: 0.956910
Epoch: 13 Accuracy: 0.956509
Epoch: 14 Accuracy: 0.958012
Epoch: 15 Accuracy: 0.959014
Epoch: 16 Accuracy: 0.957411
Epoch: 17 Accuracy: 0.958413
Epoch: 18 Accuracy: 0.958413
Epoch: 19 Accuracy: 0.958413
```

```python
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)
```
structured perceptron - extended features accuracy train: 0.946
dev: 0.868 test: 0.840

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.  

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 $P(y' \mid x)$ for a given variable, it finds the posterior marginals for all factors (nodes and edges), $P(\bar{y} \mid \bar{x})$ and $P(\bar{y}, \bar{y}_{-1} \mid \bar{x})$. 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.

Exercise 3.3 Repeat Exercises 3.1–3.2 using a CRF model instead of the perceptron algorithm. Report the results.
Algorithm 11 Batch Gradient Descent for Conditional Random Fields

1: input: $D$, $\lambda$, number of rounds $T$, learning rate sequence $(\eta_t)_{t=1,...,T}$
2: initialize $\theta^1 = 0$
3: for $t = 1$ to $T$ do
4:     for $m = 1$ to $M$ do
5:         take training pair $(x^m, y^m)$ and compute conditional probabilities using the current model, for each $\tilde{y}$:
6:         \[
6:         P_{\theta_t}(\tilde{y}|\tilde{x}) = \frac{1}{Z(\theta^t, \tilde{x})} \exp \left( \sum_i \theta^t_V \cdot f_V(\tilde{x}, y_i) + \sum_i \theta^t_E \cdot f_E(\tilde{x}, y_i, y_{i-1}) \right)
6:         \]
7:         compute the feature vector expectation:
8:         \[
8:         E_{\theta_t}[f(\tilde{x}^m, \tilde{y}^m)] = \sum_{\tilde{y}} P_{\theta_t}(\tilde{y}^m|\tilde{x}^m) f(\tilde{x}^m, \tilde{y}^m)
8:         \]
9:     end for
10: choose the stepsize $\eta_t$ using, e.g., Armijo’s rule
11: update the model:
12: \[
12: \theta^{t+1} \leftarrow (1 - \lambda \eta_t) \theta^t + \eta_t M^{-1} \sum_{m=1}^M (f(\tilde{x}^m, \tilde{y}^m) - E_{\theta_t}[f(\tilde{x}^m, \tilde{y}^m)])
12: \]
13: end for
14: output: $\hat{\theta} \leftarrow \theta^{T+1}$

```
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)
```
Here is the code for the extended feature set:

```python
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