In this class we will address the problem of unsupervised learning of linguistic structures, namely parts-of-speech. In this setting we are not given any labeled data. Instead, all we get to see is a set of natural language sentences. The underlying question is:

Can we learn something from raw text?

This task is particularly challenging since the process by which linguistic structures are generated is not always clear and even when it is, it is normally too complex to be formally expressed. Nevertheless, unsupervised learning has been applied to a wide range of natural language processing tasks, such as: Part-of-Speech Induction (Schütze 1995; Merialdo 1994; Clark 2003), Dependency Grammar Induction (Klein and Manning 2004; Smith and Eisner 2006), Constituency Grammar Induction (Klein and Manning 2004), Statistical Word Alignments (Brown et al. 1993) and Anaphora Resolution (Charniak and Elsner 2009), just to name a few.

Different motivations have pushed research in this area. From both a linguistic and cognitive point of view, unsupervised learning is useful as a tool to study language acquisition. From a machine learning point of view, unsupervised learning is a fertile ground for testing new learning methods, where significant improvements can yet be made. From a more pragmatic perspective, unsupervised learning is required since annotated corpora is a scarce resource for different reasons. Independently of the reason, unsupervised learning is an increasing active field of research.

A first problem with unsupervised learning, since we don’t observe any labeled data (i.e., the training set is now $D = \{x_1, \ldots, x_M\}$), is that most of the methods studied so far (Perceptron, Mira, SVMs) cannot be used since we cannot compare the true output with the predicted output. Note also that a direct minimization of the complete negative log-likelihood of the data, $\log P_\theta(D)$,
is very challenging, since it would require marginalizing out (i.e., summing over) all possible hidden variables:

$$\log P_\theta(D) = \sum_{m=1}^{M} \log \sum_{y \in Y} P_\theta(x_m, y).$$  \hfill (5.1)

Note also that the objective above is non-convex even for a linear model: hence, it may have local minima, which makes optimization much more difficult.

Another observation is that normally we are restricted to generative models, with some remarkable exceptions [Smith and Eisner 2005a], since the objective of discriminative models when no labels are observed are meaningless ($\sum_{y_m} P(y_m|x^m) = 1$); this rules out, for instance, Maximum Entropy classifiers.

The most common optimization method in the presence of hidden (latent) variables is the Expectation Maximization (EM) algorithm. Note that this algorithm is a generic optimization routine that does not depend on a particular model. The next section will explain the EM algorithm. On Section 5.2, we will apply the EM algorithm to the task of part-of-speech induction, where one is given raw text and a number of clusters and the task is to cluster words that behave similarly in a grammatical sense.

### 5.1 Expectation Maximization Algorithm

Given a particular model $p_\theta(\tilde{x}, \tilde{y})$ and a training corpus $\mathcal{X}$ of $D$ sentences $\tilde{x}^1 \ldots \tilde{x}^D$, training seeks model parameters $\theta$ that minimize the negative log-likelihood of the corpus:

$$\text{Negative Log Likelihood: } \mathcal{L}(\theta) = \hat{E}[-\log p_\theta(\tilde{x})] = \hat{E}[-\log \sum_{\tilde{y}} p_\theta(\tilde{x}, \tilde{y})],$$  \hfill (5.2)

where $\hat{E}[f(\tilde{x})] = \frac{1}{D} \sum_{d=1}^{D} f(\tilde{x}^d)$ denotes the empirical average of a function $f$ over the training corpus.

Because of the hidden variables $\tilde{y}$, the likelihood term contains a sum over all possible hidden structures inside of a logarithm, which makes this quantity hard to compute.

The most common minimization algorithm to fit the model parameters in the presence of hidden variables is the Expectation Maximization (EM) algorithm.

The EM procedure can be thought of intuitively in the following way. If we observe the hidden variables’ values for all sentences in the corpus, then we could easily compute the maximum likelihood value of the parameters as described in Section 2.2. On the other hand, if we had the model parameters we could label data using the model, and collect the sufficient statistics described in Section 2.2. Since we are working in an unsupervised setting, we never
get to observe the hidden state sequence. Instead, given a training set \( X = \{ \bar{x}^1 \ldots \bar{x}^D \} \), we will need to collect sufficient statistics, or expected counts that represent the expected number of times that each hidden variable is expected to be used with the current parameters setting. These sufficient statistics will then be used during learning as fake observations of the hidden variables. Using the node and edge posterior distributions described in Equations 2.17 and 2.18, the sufficient statistics can be computed by the following formulas:

\[
\text{Initial Counts: } i_c(y_l) = \sum_{d=1}^{D} \gamma_1(y_l); \\
\text{Final Counts: } f_c(y_N, y_{N-1}) = \sum_{d=1}^{D} \xi_{N-1}(y_l, y_m); \\
\text{Transition Counts: } t_c(y_l, y_m) = \sum_{d=1}^{D} \sum_{i=1}^{N-1} \xi_i(y_l, y_m); \\
\text{State Counts: } s_c(v_q, y_m) = \sum_{d=1}^{D} \sum_{i=1}^{N} \gamma_i(y_m). \]

Compare the previous Equations with the ones described in Section 2.2 for the same quantities. The main difference is that while in the presence of supervised data you sum the observed events, when you have no label data you sum the posterior probabilities of each event. If these probabilities were such that the probability mass was around single events then both Equations will produce the same result.

The EM procedure starts with an initial guess for the parameters \( \theta^0 \) at time \( t = 0 \). The algorithm iterates for \( T \) iterations until it converges to a local minima of the negative log likelihood, and each iteration is divided into two steps:

The first step - “E Step” (Expectation) - computes the posteriors for the hidden variables \( p_\theta(\bar{y} | \bar{x}) \), given the current parameter values \( \theta^t \) and the observed variables. In the case of the HMM this requires only to run the FB algorithm.

The second step - “M step” (Maximization) - uses \( p_\theta(\bar{y} | \bar{x}) \) to “softly fill in” the values of the hidden variables \( \bar{y} \), and collects the sufficient statistics, initial counts (Eq: 5.3), transition counts (Eq: 5.5) and state counts (Eq: 5.6) and uses those counts to estimate maximum likelihood parameters \( \theta^{t+1} \) as described in Section 2.2.

The EM algorithm is guaranteed to converge to a local minimum of \( \mathcal{L}(\theta) \) under mild conditions. Note that we are not committing to the best assignment of the hidden variables, but summing the occurrences of each parameter weighed by the posterior probability of all possible assignments. This modular
split into two intuitive and straightforward steps accounts for the vast popularity of EM.

More formally, EM minimizes $\mathcal{L}(\theta)$ via block-coordinate descent on an upper bound $F(q, \theta)$ using an auxiliary distribution over the latent variables $q(\bar{y} \mid \bar{x})$:

$$L(\theta) = \hat{E} \left[ - \log \sum_{\bar{y}} p_\theta(\bar{x}, \bar{y}) \right] \quad (5.7)$$

$$= \hat{E} \left[ - \log \sum_{\bar{y}} q(\bar{y} \mid \bar{x}) * \frac{p_\theta(\bar{x}, \bar{y})}{q(\bar{y} \mid \bar{x})} \right] \leq \hat{E} \left[ - \sum_{\bar{y}} q(\bar{y} \mid \bar{x}) \log \frac{p_\theta(\bar{x}, \bar{y})}{q(\bar{y} \mid \bar{x})} \right] \quad (5.8)$$

$$= \hat{E} \left[ \sum_{\bar{y}} q(\bar{y} \mid \bar{x}) \log \frac{q(\bar{y} \mid \bar{x})}{p_\theta(\bar{x}, \bar{y})} \right] = F(q, \theta), \quad (5.9)$$

where we have multiplied and divided the $p_\theta(\bar{x}, \bar{y})$ by the same quantity $q(\bar{y} \mid \bar{x})$, and the lower bound comes from applying Jensen Inequality (Equation 5.8). $F(q, \theta)$ is normally referred to as the energy function, which comes from the physics field and refers to the energy of a given system that we want to minimize.

**EM Upper Bound:**

$$\mathcal{L}(\theta) \leq F(q, \theta) = \hat{E} \left[ \sum_{\bar{y}} q(\bar{y} \mid \bar{x}) \log \frac{q(\bar{y} \mid \bar{x})}{p_\theta(\bar{x}, \bar{y})} \right]. \quad (5.10)$$

The alternating E and M steps at iteration $t + 1$ can be seen as minimizing the energy function first with respect to $q(\bar{y} \mid \bar{x})$ and then with respect to $\theta$:

**E:**

$$q^{t+1}(\bar{y} \mid \bar{x}) = \arg \min_{q(\bar{y} \mid \bar{x})} F(q, \theta^t) = \arg \min_{q(\bar{y} \mid \bar{x})} \text{KL}(q(\bar{y} \mid \bar{x}) \| p_\theta(\bar{y} \mid \bar{x})) = p_{\theta^t}(\bar{y} \mid \bar{x}) \quad (5.11)$$

**M:**

$$\theta^{t+1} = \arg \min_{\theta} F(q^{t+1}, \theta) = \arg \max_{\theta} \hat{E} \left[ \sum_{\bar{y}} q^{t+1}(\bar{y} \mid \bar{x}) \log p_\theta(\bar{x}, \bar{y}) \right]; \quad (5.12)$$

where $\text{KL}(q \| p) = \mathbb{E}_q[\log \frac{q(\cdot)}{p(\cdot)}]$ is the Kullback-Leibler divergence. The KL term in the E-Step results from dropping all terms from the energy function that are constant for a set $\theta$, in this case the likelihood of the observation sequence $p_\theta(\bar{x})$:
\[
\sum_y q(\bar{y} | \bar{x}) \log \frac{q(\bar{y} | \bar{x})}{p_\theta(\bar{y}, \bar{x})} = \sum_y q(\bar{y} | \bar{x}) \log q(\bar{y} | \bar{x}) - \sum_y q(\bar{y} | \bar{x}) \log p_\theta(\bar{x}, \bar{y}) \\
= \sum_y q(\bar{y} | \bar{x}) \log q(\bar{y} | \bar{x}) - \sum_y q(\bar{y} | \bar{x}) \log p_\theta(\bar{x}) p_\theta(\bar{y} | \bar{x}) \\
= \sum_y q(\bar{y} | \bar{x}) \log \frac{q(\bar{y} | \bar{x})}{p_\theta(\bar{y} | \bar{x})} - \log p_\theta(\bar{x}) \\
= KL(q(\bar{y} | \bar{x}) || p_\theta(\bar{y} | \bar{x})) - \log p_\theta(\bar{x}).
\]

Algorithm 13 presents the pseudo code for the EM algorithm. Note that this algorithm is agnostic of a particular model, it only requires the model to implement a common interface.

Algorithm 13 EM algorithm.

```python
1: input: dataset D, an initialized model
2: for t = 1 to T do
3:   model.clear_counts()
4:   for seq ∈ D do
5:     E-Step:
6:     posteriors, likelihood = model.compute_posteriors(seq)
7:     model.update_counts(seq, posteriors)
8:   end for
9:   M-Step:
10:  model.update_params(counts)
11: end for
```

One important thing to note in Algorithm 13 is that for the HMM model we already have all the model pieces we require. In fact the only method we don’t have yet implemented from previous classes is the method to update_counts(posteriors).

Exercise 5.1 Implement the method update_counts(seq, posteriors).

```python
1 def update_counts(self, seq, posteriors):
```

Use the method you defined previously to check the count tables to check if this method is correct. Use a corpus with only one sentence to make the test simpler.

```python
In [ ]: run readers/pos_corpus.py
In [ ]: posc = PostagCorpus("en", max_sent_len=15, train_sents=1, dev_sents=0, test_sents=0)
```
In []: run sequences/hmm.py
In []: hmm = HMM(posc)
In []: hmm.train_supervised(posc.train, smoothing=0.1)
In []: hmm.clear_counts()
In []: posteriors, likelihood = hmm.get_posteriors(posc.train.
    seq_list[0])
In []: hmm.update_counts(posc.train.seq_list[0], posteriors)
In []: hmm.sanity_check_counts(posc.train)

If you pass this test, then you have all the pieces to implement the EM algorithm.
Look at the code for EM algorithm in file sequences/em.py and check it for yourself.

def train(self, seq_list, nr_iter=10, smoothing=0, evaluate=True):
    if (evaluate):
        ### Evaluate accuracy at initial iteration
        pred = self.model.viterbi_decode_corpus(seq_list.
            seq_list)
        acc = self.model.evaluate_corpus(seq_list.seq_list,
            pred)
        for t in xrange(1, nr_iter):
            # E-Step
            total_likelihood = 0
            self.model.clear_counts(smoothing)
            for seq in seq_list.seq_list:
                posteriors, likelihood = self.model.
                    get_posteriors(seq)
                self.model.update_counts(seq, posteriors)
                total_likelihood += likelihood
                print "Iter: %i - Log Likelihood %f"%(t,-1*math.log
                    (total_likelihood))
                # M-Step
                self.model.update_params()
    if (evaluate):
        ### Evaluate accuracy at this iteration
        pred = self.model.viterbi_decode_corpus(
            seq_list.seq_list)
        acc = self.model.evaluate_corpus(seq_list.
            seq_list, pred)
        print "Iter: %i acc %f"%(t, acc)
5.2 Part of Speech Induction

In this section we present the Part-of-Speech induction task. Part-of-Speech tags are pre-requisite for many text applications. The task of Part-of-Speech tagging where one is given a labeled training set of words and respective tags is a well studied task with several methods achieving high prediction quality, as we saw in Chapters 2 and 3.

On the other hand the task of Part-of-Speech induction where one does not have access to a labeled corpus is a much harder task with a huge space for improvement. In this case, we are given only the raw text along with sentence boundaries and a predefined number of clusters we can use. This problem can be seen as a clustering problem. We want to cluster words that behave grammatically in the same way on the same cluster. This is a much harder problem.

Formally, the problem setting is the following: we are given a training set \( \mathcal{X} = \mathbf{x}_1, \ldots, \mathbf{x}_D \) of \( D \) training examples, where each example \( \mathbf{x} = x_1, \ldots, x_N \) is a sentence of \( N \) words, whose values \( v \) are taken from a vocabulary \( \mathcal{V} \) of possible word types. We are also given the set of clusters \( \mathcal{Y} \) that we are allowed to use. The hidden structure \( \mathbf{y} = y_1, \ldots, y_N \) corresponds to a sequence of cluster assignments for each individual word, such that \( y_n = y_l \) with \( y_l \in \mathcal{Y} \).

Depending on the task at hand we can pick an arbitrary number of clusters. If the goal is to test how well our method can recover the true pos tags then we should use the same number of clusters as pos tags. On the other hand, if the task is to extract features to be used by other methods we can use a much bigger number of clusters (e.g. 200) to capture correlations not captured by pos tags, like lexical affinity.

Note, however that nothing is said about the identity of each cluster. The model has no preference in assigning cluster 1 to nouns vs cluster 2 to nouns. Given this non-identifiability several metrics have been proposed for evaluation (Reichart and Rappoport, 2009; Haghighi and Klein, 2006; Meila, 2007; Rosenberg and Hirschberg, 2007). In this class we will use a common and simple metric called 1-Many, which maps each cluster to majority pos tag that it contains (see Figure 5.1 for an example).

**Exercise 5.2 Run the EM algorithm for part of speech induction:**

```python
In [1]: run readers/pos_corpus.py
In [1]: posc = PostagCorpus("en", max_sent_len=15, train_sents=1000, dev_sents=0, test_sents=0)
In [1]: run sequences/hmm.py
In [1]: hmm = HMM(posc)
In [1]: hmm.initialize_random()
In [1]: run sequences/em.py
In [1]: em = EM(posc, hmm)
In [1]: em.train(posc.train, nr_iter=20)
```
Figure 5.1: Confusion Matrix example. Each cluster is a column. The best tag in each column is represented under the column (1-many) mapping. Each color represents a true Pos Tag.
In the previous exercise we used an HMM to do Part-of-Speech induction using 12 clusters (by omission the HMM uses as number of hidden states the one provided by the corpus). A first observation is that the log-likelihood is always increasing as expected. Another observation is that the accuracy goes up from 33% to 41%. Note that normally you will run this algorithm for 200 iterations, we stopped earlier for time constraints. Another observations is that the accuracy is not monotonic increasing, this is because the likelihood is not a perfect proxy for the accuracy. In fact all that likelihood is measuring are co-occurrences of words in the corpus; it has no idea of pos tags. The fact we are improving derives from the fact that language is not random but follows some specific hidden patterns. In fact this patterns are what true pos-tags try to capture. A final observation is that the performance is really bad compared to the supervised scenario, so there is a lot of space for improvement. The actual state of the art is around 71% for fully unsupervised (Graça 2010; Berg-Kirkpatrick et al. 2010) and 80% (Das and Petrov 2011) using parallel data and information from labels in the other language.

Looking at Figure 5.1 shows the confusion matrix for this particular example. A first observation is that most clusters are mapped to nouns, verbs or punctuation. This is a none fact since there are many more nouns and verbs than any other tags. Since maximum likelihood prefers probabilities to be uniform (Imagine two parameters. In one setting both have value 0.5 so the likelihood will be 0.5*0.5 = 0.25, while in the other case one as 0.1 and 0.9 so the maximum likelihood is 0.09). Several approaches have been proposed to address this problem under moving towards a Bayesian setting or using Posterior
Regularization (Johnson 2007; Graça et al. 2009) more about this later today.
Part-of-Speech induction is a very active field of research, in fact in the last two
ACL conferences (Association for Computational Linguistics) the short paper
award (2010) and the best paper award (2011) were about this topic (Lamar
et al. 2010; Das and Petrov 2011).
Bibliography


