16 | Expectation Maximization

A hen is only an egg's way of making another egg. – Samuel Butler

Learning Objectives:

- Explain the relationship between parameters and hidden variables.
- Construct generative stories for clustering and dimensionality reduction.
- Draw a graph explaining how EM works by constructing convex lower bounds.
- Implement EM for clustering with mixtures of Gaussians, and contrasting it with k-means.
- Evaluate the differences betweem EM and gradient descent for hidden variable models.

Dependencies:

Suppose you were building a naive Bayes model for a text categorization problem. After you were done, your boss told you that it became prohibitively expensive to obtain labeled data. You now have a probabilistic model that assumes access to labels, but you don't have any labels! Can you still do something?

Amazingly, you can. You can treat the labels as **hidden variables**, and attempt to learn them at the same time as you learn the parameters of your model. A very broad family of algorithms for solving problems just like this is the **expectation maximization** family. In this chapter, you will derive expectation maximization (EM) algorithms for clustering and dimensionality reduction, and then see why EM works.

16.1 Grading an Exam without an Answer Key

Alice's machine learning professor Carlos gives out an exam that consists of 50 true/false questions. Alice's class of 100 students takes the exam and Carlos goes to grade their solutions. If Carlos made an answer key, this would be easy: he would just count the fraction of correctly answered questions each student got, and that would be their score. But, like many professors, Carlos was really busy and didn't have time to make an answer key. Can he still grade the exam?

There are two insights that suggest that he might be able to. Suppose he know ahead of time that Alice was an awesome student, and is basically guaranteed to get 100% on the exam. In that case, Carlos can simply use Alice's answers as the ground truth. More generally, if Carlos assumes that *on average* students are better than random guessing, he can hope that the majority answer for each question is likely to be correct. Combining this with the previous insight, when doing the "voting", he might want to pay more attention to the answers of the better students.

To be a bit more pedantic, suppose there are $N = 100$ students and $M = 50$ questions. Each student n has a score s_n, between 0 and

1 that denotes how well they do on the exam. The score is what we really want to compute. For each question m and each student n, the student has provided an answer $a_{n,m}$, which is either zero or one. There is also an unknown ground truth answer for each question m, which we'll call t_m, which is also either zero or one.

As a starting point, let's consider a simple heuristic and then complexify it. The heuristic is the "majority vote" heuristic and works as follows. First, we estimate t_m as the most common answer for question m: $t_m = \text{argmax}_t \sum_n \mathbf{1}[a_{n,m} = t]$. Once we have a guess for each true answer, we estimate each students' score as how many answers they produced that match this guessed key: $s_n = \frac{1}{M} \sum_m \mathbf{1}[a_{n,m} = t_m]$.

Once we have these scores, however, we might want to trust some of the students more than others. In particular, answers from students with high scores are perhaps more likely to be correct, so we can *recompute* the ground truth, according to weighted votes. The weight of the votes will be precisely the score the corresponding each student:

$$t_m = \operatorname*{argmax}_t \sum_n s_n \mathbf{1}[a_{n,m} = t] \tag{16.1}$$

You can recognize this as a *chicken and egg problem*. If you knew the student's scores, you could estimate an answer key. If you had an answer key, you could compute student scores. A very common strategy in computer science for dealing with such chicken and egg problems is to iterate. Take a guess at the first, compute the second, recompute the first, and so on.

In order to develop this idea formally, we have to case the problem in terms of a probabilistic model with a generative story. The generative story we'll use is:

1. For each question m, choose a true answer $t_m \sim \mathcal{B}er(0.5)$
2. For each student n, choose a score $s_n \sim \mathcal{U}ni(0, 1)$
3. For each question m and each student n, choose an answer $a_{n,m} \sim \mathcal{B}er(s_n)^{t_m} \mathcal{B}er(1 - s_n)^{1-t_m}$

In the first step, we generate the true answers independently by flipping a fair coin. In the second step, each students' overall score is determined to be a uniform random number between zero and one. The tricky step is step three, where each students' answer is generated for each question. Consider student n answering question m, and suppose that $s_n = 0.9$. If $t_m = 1$, then $a_{n,m}$ should be 1 (i.e., correct) 90% of the time; this can be accomplished by drawing the answer from $\mathcal{B}er(0.9)$. On the other hand, if $t_m = 0$, then $a_{n,m}$ should 1 (i.e., incorrect) 10% of the time; this can be accomplished by drawing

the answer from $\mathcal{B}er(0.1)$. The exponent in step 3 selects which of two Bernoulli distributions to draw from, and then implements this rule.

This can be translated into the following likelihood:

$$
\begin{aligned}
& p(\boldsymbol{a}, \boldsymbol{t}, \boldsymbol{s}) \\
&= \left[\prod_m 0.5^{t_m} 0.5^{1-t_m}\right] \times \left[\prod_n 1\right] \\
&\quad \times \left[\prod_n \prod_m s_n^{a_{n,m} t_m} (1-s_n)^{(1-a_{n,m}) t_m} \right. \\
&\qquad \left. s_n^{(1-a_{n,m})(1-t_m)} (1-s_n)^{a_{n,m}(1-t_m)}\right] && (16.2) \\
&= 0.5^M \prod_n \prod_m s_n^{a_{n,m} t_m} (1-s_n)^{(1-a_{n,m}) t_m} s_n^{(1-a_{n,m})(1-t_m)} (1-s_n)^{a_{n,m}(1-t_m)} && (16.3)
\end{aligned}
$$

Suppose we knew the true lables $\boldsymbol{t}$. We can take the log of this likelihood and differentiate it with respect to the score s_n of some student (note: we can drop the 0.5^M term because it is just a constant):

$$
\begin{aligned}
\log p(\boldsymbol{a}, \boldsymbol{t}, \boldsymbol{s}) &= \sum_n \sum_m \Big[a_{n,m} t_m \log s_n + (1-a_{n,m})(1-t_m) \log(s_n) \\
&\qquad + (1-a_{n,m}) t_m \log(1-s_n) + a_{n,m}(1-t_m) \log(1-s_n)\Big] && (16.4) \\
\frac{\partial \log p(\boldsymbol{a}, \boldsymbol{t}, \boldsymbol{s})}{\partial s_n} &= \sum_m \left[\frac{a_{n,m} t_m + (1-a_{n,m})(1-t_m)}{s_n} - \frac{(1-a_{n,m}) t_m + a_{n,m}(1-t_m)}{1-s_n}\right] && (16.5)
\end{aligned}
$$

The derivative has the form $\frac{A}{s_n} - \frac{B}{1-s_n}$. If we set this equal to zero and solve for s_n, we get an optimum of $s_n = \frac{A}{A+B}$. In this case:

$$
\begin{aligned}
A &= \sum_m \left[a_{n,m} t_m + (1-a_{n,m})(1-t_m)\right] && (16.6) \\
B &= \sum_m \left[(1-a_{n,m}) t_m + a_{n,m}(1-t_m)\right] && (16.7) \\
A + B &= \sum_m [1] = M && (16.8)
\end{aligned}
$$

Putting this together, we get:

$$
s_n = \frac{1}{M} \sum_m \left[a_{n,m} t_m + (1-a_{n,m})(1-t_m)\right] \tag{16.9}
$$

In the case of known ts, this matches exactly what we had in the heuristic.

However, we do not know t, so instead of using the "true" values of t, we're going to use their *expectations*. In particular, we will compute s_n by maximizing its likelihood under the *expected* values

of t, hence the name **expectation maximization**. If we are going to compute expectations of t, we have to say: expectations according to which probability distribution? We will use the distribution $p(t_m \mid \boldsymbol{a}, \boldsymbol{s})$. Let $\tilde{t}_m$ denote $\mathbb{E}_{t_m \sim p(t_m \mid \boldsymbol{a}, \boldsymbol{s})}[t_m]$. Because t_m is a binary variable, its expectation is equal to it's probability; namely: $\tilde{t}_m = p(t_m \mid \boldsymbol{a}, \boldsymbol{s})$.

How can we compute this? We will compute $C = p(t_m = 1, \boldsymbol{a}, \boldsymbol{s})$ and $D = p(t_m = 0, \boldsymbol{a}, \boldsymbol{s})$ and then compute $\tilde{t}_m = C/(C + D)$. The computation is straightforward:

$$C = 0.5 \prod_n s_n^{a_{n,m}} (1 - s_n)^{1 - a_{n,m}} = 0.5 \prod_{\substack{n: \\ a_{n,m}=1}} s_n \prod_{\substack{n: \\ a_{n,m}=0}} (1 - s_n) \quad (16.10)$$

$$D = 0.5 \prod_n s_n^{1 - a_{n,m}} (1 - s_n)^{a_{n,m}} = 0.5 \prod_{\substack{n: \\ a_{n,m}=1}} (1 - s_n) \prod_{\substack{n: \\ a_{n,m}=0}} s_n \quad (16.11)$$

If you inspect the value of C, it is basically "voting" (in a product form, not a sum form) the scores of those students who agree that the answer is 1 with one-minus-the-score of those students who do not. The value of D is doing the reverse. This is a form of multiplicative voting, which has the effect that if a given student has a perfect score of 1.0, their results will carry the vote completely.

We now have a way to:

1. Compute expected ground truth values $\tilde{t}_m$, given scores.
2. Optimize scores s_n given expected ground truth values.

The full solution is then to alternate between these two. You can start by initializing the ground truth values at the majority vote (this seems like a safe initialization). Given those, compute new scores. Given those new scores, compute new ground truth values. And repeat until tired.

In the next two sections, we will consider a more complex unsupervised learning model for clustering, and then a generic mathematical framework for expectation maximization, which will answer questions like: will this process converge, and, if so, to what?

16.2 *Clustering with a Mixture of Gaussians*

In Chapter 9, you learned about probabilitic models for classification based on density estimation. Let's start with a fairly simple classification model that *assumes* we have labeled data. We will shortly remove this assumption. Our model will state that we have K classes, and data from class k is drawn from a Gaussian with mean $\boldsymbol{\mu}_k$ and variance σ_k^2. The choice of classes is parameterized by $\boldsymbol{\theta}$. The generative story for this model is:

1. For each example $n = 1 \dots N$:
   (a) Choose a label $y_n \sim \mathcal{D}isc(\boldsymbol{\theta})$
   (b) Choose example $\boldsymbol{x}_n \sim \mathcal{N}or(\boldsymbol{\mu}_{y_n}, \sigma^2_{y_n})$

This generative story can be directly translated into a likelihood as before:

$$p(D) = \prod_n \mathcal{M}ult(y_n \mid \boldsymbol{\theta}) \mathcal{N}or(\boldsymbol{x}_n \mid \boldsymbol{\mu}_{y_n}, \sigma^2_{y_n}) \tag{16.12}$$

$$= \overbrace{\prod_n \underbrace{\theta_{y_n}}_{\text{choose label}} \underbrace{\left[2\pi\sigma^2_{y_n}\right]^{-\frac{D}{2}} \exp\left[-\frac{1}{2\sigma^2_{y_n}} \left|\left|\boldsymbol{x}_n - \boldsymbol{\mu}_{y_n}\right|\right|^2\right]}_{\text{choose feature values}}}^{\text{for each example}} \tag{16.13}$$

If you had access to labels, this would be all well and good, and you could obtain closed form solutions for the maximum likelihood estimates of all parameters by taking a log and then taking gradients of the log likelihood:

$$\theta_k = \text{fraction of training examples in class } k = \frac{1}{N}\sum_n [y_n = k] \tag{16.14}$$

$$\boldsymbol{\mu}_k = \text{mean of training examples in class } k = \frac{\sum_n [y_n = k]\boldsymbol{x}_n}{\sum_n [y_n = k]} \tag{16.15}$$

$$\sigma^2_k = \text{variance of training examples in class } k = \frac{\sum_n [y_n = k]\, ||\boldsymbol{x}_n - \mu_k||}{\sum_n [y_n = k]} \tag{16.16}$$

Suppose that you *don't* have labels. Analogously to the K-means algorithm, one potential solution is to iterate. You can start off with guesses for the values of the unknown variables, and then iteratively improve them over time. In K-means, the approach was the *assign* examples to labels (or clusters). This time, instead of making hard assignments ("example 10 belongs to cluster 4"), we'll make **soft assignments** ("example 10 belongs half to cluster 4, a quarter to cluster 2 and a quarter to cluster 5"). So as not to confuse ourselves too much, we'll introduce a new variable, $\boldsymbol{z}_n = \langle z_{n,1}, \dots, z_{n,K}$ (that sums to one), to denote a fractional assignment of examples to clusters.

This notion of soft-assignments is visualized in Figure 16.1. Here, we've depicted each example as a pie chart, and it's coloring denotes the degree to which it's been assigned to each (of three) clusters. The size of the pie pieces correspond to the z_n values.

? You should be able to derive the maximum likelihood solution results formally by now.

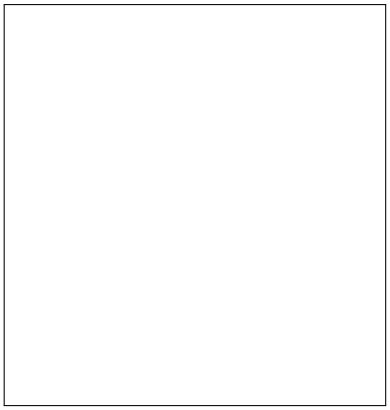

Figure 16.1: em:piecharts: A figure showing pie charts

Formally, $z_{n,k}$ denotes the probability that example n is assigned to cluster k:

$$z_{n,k} = p(y_n = k \mid \boldsymbol{x}_n) \tag{16.17}$$
$$= \frac{p(y_n = k, \boldsymbol{x}_n)}{p(\boldsymbol{x}_n)} \tag{16.18}$$
$$= \frac{1}{Z_n} \mathcal{M}ult(k \mid \boldsymbol{\theta}) \mathcal{N}or(\boldsymbol{x}_n \mid \boldsymbol{\mu}_k, \sigma_k^2) \tag{16.19}$$

Here, the normalizer Z_n is to ensure that $\boldsymbol{z}_n$ sums to one.

Given a set of parameters (the $\boldsymbol{\theta}$s, $\boldsymbol{\mu}$s and σ^2s), the **fractional assignments** $z_{n,k}$ are easy to compute. Now, akin to K-means, given fractional assignments, you need to recompute estimates of the model parameters. In analogy to the maximum likelihood solution (Eqs (**??**)-(**??**)), you can do this by counting fractional points rather than full points. This gives the following re-estimation updates:

$$\theta_k = \text{fraction of training examples in class } k \tag{16.20}$$
$$= \frac{1}{N} \sum_n z_{n,k}$$
$$\boldsymbol{\mu}_k = \text{mean of fractional examples in class } k \tag{16.21}$$
$$= \frac{\sum_n z_{n,k} \boldsymbol{x}_n}{\sum_n z_{n,k}}$$
$$\sigma_k^2 = \text{variance of fractional examples in class } k \tag{16.22}$$
$$= \frac{\sum_n z_{n,k} ||\boldsymbol{x}_n - \mu_k||}{\sum_n z_{n,k}}$$

All that has happened here is that the hard assignments "$[y_n = k]$" have been replaced with soft assignments "$z_{n,k}$". As a bit of foreshadowing of what is to come, what we've done is essentially replace known labels with *expected labels*, hence the name "expectation maximization."

Putting this together yields Algorithm 16.2. This is the **GMM** ("**Gaussian Mixture Models**") algorithm, because the probabilitic model being learned describes a dataset as being drawn from a mixture distribution, where each component of this distribution is a Gaussian.

Just as in the K-means algorithm, this approach is succeptible to local optima and quality of initialization. The heuristics for computing better initializers for K-means are also useful here.

? Aside from the fact that GMMs use soft assignments and K-means uses hard assignments, there are other differences between the two approaches. What are they?

16.3 The Expectation Maximization Framework

At this point, you've seen a method for learning in a particular probabilistic model with hidden variables. Two questions remain: (1) can

Algorithm 38 GMM($\mathbf{X}$, K)

for $k = 1$ **to** K **do**
  $\boldsymbol{\mu}_k \leftarrow$ some random location // randomly initialize mean for kth cluster
  $\sigma_k^2 \leftarrow 1$ // initialize variances
  $\theta_k \leftarrow 1/K$ // each cluster equally likely a priori
end for
repeat
  for $n = 1$ **to** N **do**
    for $k = 1$ **to** K **do**
      $z_{n,k} \leftarrow \theta_k \left[2\pi\sigma_k^2\right]^{-\frac{D}{2}} \exp\left[-\frac{1}{2\sigma_k^2} ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2\right]$ // compute (unnormalized) fractional assignments
    end for
    $\boldsymbol{z}_n \leftarrow \frac{1}{\sum_k z_{n,k}} \boldsymbol{z}_n$ // normalize fractional assignments
  end for
  for $k = 1$ **to** K **do**
    $\theta_k \leftarrow \frac{1}{N} \sum_n z_{n,k}$ // re-estimate prior probability of cluster k
    $\boldsymbol{\mu}_k \leftarrow \frac{\sum_n z_{n,k} \boldsymbol{x}_n}{\sum_n z_{n,k}}$ // re-estimate mean of cluster k
    $\sigma_k^2 \leftarrow \frac{\sum_n z_{n,k} ||\boldsymbol{x}_n - \mu_k||}{\sum_n z_{n,k}}$ // re-estimate variance of cluster k
  end for
until converged
return $\boldsymbol{z}$ // return cluster assignments

you apply this idea more generally and (2) why is it even a reasonable thing to do? Expectation maximization is a *family* of algorithms for performing maximum likelihood estimation in probabilistic models with hidden variables.

The general flavor of how we will proceed is as follows. We want to maximize the log likelihood $\mathcal{L}$, but this will turn out to be difficult to do directly. Instead, we'll pick a surrogate function $\tilde{\mathcal{L}}$ that's a lower bound on $\mathcal{L}$ (i.e., $\tilde{\mathcal{L}} \leq \mathcal{L}$ everywhere) that's (hopefully) easier to maximize. We'll construct the surrogate in such a way that increasing it will force the true likelihood to also go up. After maximizing $\tilde{\mathcal{L}}$, we'll construct a *new* lower bound and optimize that. This process is shown pictorially in Figure 16.2.

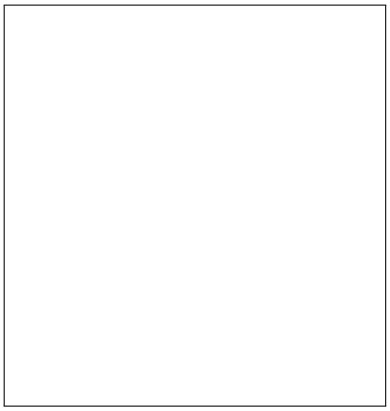

Figure 16.2: em:lowerbound: A figure showing successive lower bounds

To proceed, consider an arbitrary probabilistic model $p(\boldsymbol{x}, \boldsymbol{y} \mid \boldsymbol{\theta})$, where $\boldsymbol{x}$ denotes the observed data, $\boldsymbol{y}$ denotes the hidden data and $\boldsymbol{\theta}$ denotes the parameters. In the case of Gaussian Mixture Models, $\boldsymbol{x}$ was the data points, $\boldsymbol{y}$ was the (unknown) labels and $\boldsymbol{\theta}$ included the cluster prior probabilities, the cluster means and the cluster variances. Now, given access *only* to a number of examples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, you would like to estimate the parameters ($\boldsymbol{\theta}$) of the model.

Probabilistically, this means that some of the variables are unknown and therefore you need to marginalize (or sum) over their possible values. Now, your data consists only of $\mathbf{X} = \langle \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N \rangle$,

not the $(\boldsymbol{x}, y)$ pairs in D. You can then write the likelihood as:

$$p(\mathbf{X} \mid \boldsymbol{\theta}) = \sum_{y_1} \sum_{y_2} \cdots \sum_{y_N} p(\mathbf{X}, y_1, y_2, \ldots y_N \mid \boldsymbol{\theta}) \qquad \text{marginalization} \tag{16.23}$$

$$= \sum_{y_1} \sum_{y_2} \cdots \sum_{y_N} \prod_n p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) \qquad \text{examples are independent} \tag{16.24}$$

$$= \prod_n \sum_{y_n} p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) \qquad \text{algebra} \tag{16.25}$$

At this point, the natural thing to do is to take logs and then start taking gradients. However, once you start taking logs, you run into a problem: the log cannot eat the sum!

$$\mathcal{L}(\mathbf{X} \mid \boldsymbol{\theta}) = \sum_n \log \sum_{y_n} p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) \tag{16.26}$$

Namely, the log gets "stuck" outside the sum and cannot move in to decompose the rest of the likelihood term!

The next step is to apply the somewhat strange, but strangely useful, trick of multiplying by 1. In particular, let $q(\cdot)$ be an arbitrary probability distribution. We will multiply the $p(\ldots)$ term above by $q(y_n)/q(y_n)$, a valid step so long as q is never zero. This leads to:

$$\mathcal{L}(\mathbf{X} \mid \boldsymbol{\theta}) = \sum_n \log \sum_{y_n} q(y_n) \frac{p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta})}{q(y_n)} \tag{16.27}$$

We will now construct a lower bound using **Jensen's inequality**. This is a very useful (and easy to prove!) result that states that $f(\sum_i \lambda_i x_i) \geq \sum_i \lambda_i f(x_i)$, so long as (a) $\lambda_i \geq 0$ for all i, (b) $\sum_i \lambda_i = 1$, and (c) f is concave. If this looks familiar, that's just because it's a direct result of the definition of **concavity**. Recall that f is concave if $f(ax + by) \geq af(x) + bf(x)$ whenever $a + b = 1$.

? Prove Jensen's inequality using the definition of concavity and induction.

You can now apply Jensen's inequality to the log likelihood by identifying the list of $q(y_n)$s as the λs, log as f (which is, indeed, concave) and each "x" as the p/q term. This yields:

$$\mathcal{L}(\mathbf{X} \mid \boldsymbol{\theta}) \geq \sum_n \sum_{y_n} q(y_n) \log \frac{p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta})}{q(y_n)} \tag{16.28}$$

$$= \sum_n \sum_{y_n} \Big[q(y_n) \log p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) - q(y_n) \log q(y_n) \Big] \tag{16.29}$$

$$\triangleq \tilde{\mathcal{L}}(\mathbf{X} \mid \boldsymbol{\theta}) \tag{16.30}$$

Note that this inequality holds for *any* choice of function q, so long as its non-negative and sums to one. In particular, it needn't even by the

same function q for each n. We will need to take advantage of both of these properties.

We have succeeded in our first goal: constructing a lower bound on $\mathcal{L}$. When you go to optimize this lower bound for $\boldsymbol{\theta}$, the only part that matters is the first term. The second term, $q \log q$, drops out as a function of $\boldsymbol{\theta}$. This means that the the maximization you need to be able to compute, for fixed q_ns, is:

$$\boldsymbol{\theta}^{(\text{new})} \leftarrow \arg\max_{\boldsymbol{\theta}} \sum_n \sum_{y_n} q_n(y_n) \log p(\boldsymbol{x}_n, y_n \mid \boldsymbol{\theta}) \tag{16.31}$$

This is *exactly* the sort of maximization done for Gaussian mixture models when we recomputed new means, variances and cluster prior probabilities.

The second question is: what should $q_n(\cdot)$ actually be? Any reasonable q will lead to a lower bound, so in order to choose one q over another, we need another criterion. Recall that we are hoping to maximize $\mathcal{L}$ by instead maximizing a lower bound. In order to ensure that an increase in the lower bound implies an increase in $\mathcal{L}$, we need to ensure that $\mathcal{L}(\mathbf{X} \mid \boldsymbol{\theta}) = \tilde{\mathcal{L}}(\mathbf{X} \mid \boldsymbol{\theta})$. In words: $\tilde{\mathcal{L}}$ should be a lower bound on $\mathcal{L}$ that makes contact at the current point, $\boldsymbol{\theta}$.

16.4 Further Reading

TODO further reading