16 Clustering

Clustering is an **unsupervised** learning problem in which our goal is to discover "clusters" in the data. A cluster is a collection of data points that are similar in some way. In other words the goal is to discover groups of data points that share similar properties.

The potential applications of clustering are remarkably widespread. A market researcher might want to identify distinct groups of the population with similar preferences and desires. When working with documents you might want to find clusters of documents based on the occurrence frequency of certain words. For example, this might allow one to discover financial documents, legal documents, or email from friends. Working with image collections you might find clusters of images which are images of people versus images of buildings. Often when we are given large amounts of complicated data we want to look for some underlying structure in the data, which might reflect certain *natural kinds* within the training data.

Clustering can be used to compress data, by replacing all of the elements in a cluster with a single representative element. Similarly, it can be used for fast approximate near neighbor search in which one takes as input a query, and then returns data points that are most similar to the query. For this one can first find the nearest clusters to the query, and then inspect the elements belonging to the nearby clusters, thereby ignoring vast amounts of data that are not in the nearby clusters.

16.1 K-Means

We begin with a simple method called K-means. Given N input data vectors $\{\mathbf{y}_i\}_{i=1}^N$, we wish to label each vector as belonging to one of K clusters. This labeling will be specified with a binary matrix $\mathbf{L}$, the elements of which are given by

$$\ell_{i,j} \;=\; \begin{cases} 1 & \text{if data point } i \text{ belongs to cluster } j \\ 0 & \text{otherwise} \end{cases} \tag{16.1}$$

We also assume that the clustering should partition the data, so each vector can only be assigned to one cluster, i.e., for all points i, $\sum_{j=1}^K \ell_{i,j} = 1$. In addition to assigning points to clusters, we also want to find a representative for each cluster, denoted $\mathbf{c}_j$. For example it could be the mean of the points assigned to that respective cluster.

The objective of K-means clustering is to cluster the points in a way that minimizes the distance between each data point and the center of the cluster to which it is assigned. One can define the K-means objective function directly in terms of the cluster centres, $\mathbf{c}_j$, as follows:

$$E(\{\mathbf{c}_j\}_{j=1..K}) \;=\; \sum_{i,j} \min_j ||\mathbf{y}_i - \mathbf{c}_j||^2 \,. \tag{16.2}$$

One can also express the objective in terms of cluster centres and the assignment matrix $\mathbf{L}$, i.e.,

$$E(\{\mathbf{c}_j\}_{j=1..K}, \mathbf{L}) \;=\; \sum_{i,j} \ell_{i,j} ||\mathbf{y}_i - \mathbf{c}_j||^2 \,, \tag{16.3}$$

subject to the constraints on the assignment variables, i.e, ℓ_{ij} is 0 or 1, and $\sum_{j=1}^K \ell_{i,j} = 1$. This objective penalizes the squared Euclidean distance between each data point and the center of the

cluster to which it is assigned. To minimize this error, we want to bring cluster centers close to the points within their clusters, and we want to assign each data point to the nearest cluster.

This optimization problem is NP-hard, and cannot be solved in closed form. It requires an iterative solution. Because it includes discrete variables (the labels $\mathbf{L}$), gradient-based methods are not straightforwardly applicable. Instead, we use a strategy called **block coordinate descent**, in which we alternate between closed-form optimization of one set of variables, holding the other variables fixed. And then, we instead hold the first set of variables fixed while we update the second.

The K-means algorithm, also known as *Lloyd's algorithm*, begins by choosing some initial values for the variables (e.g., the centers). Then we alternate two steps: 1) holding fixed the current centers, update the labels by assigning each data point to the closest center; and 2) holding fixed the current labels, update each cluster center to be the mean of all points assigned to that cluster (i.e., the LS solution). A summary of the K-means algorithm follows:

pick initial values for $\mathbf{L}$ and $\mathbf{c}_{1:K}$
loop
  // Labeling update: set $\mathbf{L} \leftarrow \arg\min_{\mathbf{L}} E(\mathbf{c}, \mathbf{L})$
  for each data point i **do**
    $j \leftarrow \arg\min_j ||\mathbf{y}_i - \mathbf{c}_j||^2$
    $\ell_{i,j} = 1$
    $\ell_{i,a} = 0$ for all $a \neq j$
  end for

  // Centers update: set $\mathbf{c} \leftarrow \arg\min_{\mathbf{c}} E(\mathbf{c}, \mathbf{L})$
  for each center j **do**
    $\mathbf{c}_j \leftarrow (\sum_i \ell_{i,j}\mathbf{y}_i) \,/\, (\sum_i \ell_{i,j})$
  end for
end loop

Importantly, each step of the optimization is guaranteed to lower the objective function until the algorithm converges (you should be able to show that each step is indeed optimal.) Nevertheless, there is no guarantee that the algorithm will find the global optimum, or even a good solution. This algorithm can easily get trapped in a poor local minima.

Initialization. The algorithm is sensitive to initialization, and poor initialization can sometimes lead to poor results. Here are a few strategies that can be used to initialize the algorithm:

1. **Random labeling:** Initialize the labeling $\mathbf{L}$ randomly, and then run the center-update step to determine the initial centers. This approach is not recommended because the initial centers will likely end up just being very close to the mean of the entire datasetset.

2. **Random initial centers:** We could place initial center locations randomly, e.g., by random sampling in the bounding box of the data. But it is very likely that some of the centers will

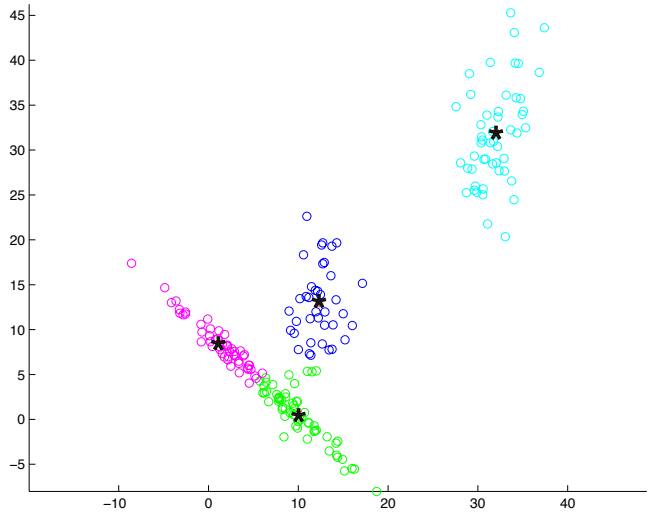

Figure 16.1: K-means applied to a dataset sampled from three Gaussian distributions. Each data assigned to one of $K = 4$ clusters. Data points are shown as circles with one of four colors. The four cluster centers are depicted by black stars.

fall into empty regions of feature space, and therefore be assigned no data. Getting a good initialization this way can be difficult.

3. **Random data points as centers:** Choosing a random subset of the datapoints as initial centers works somewhat better.

4. **Multiple restarts.** We run K-means multiple times, each time with a different random initialization (using one of the above methods). We then keep the solution found in that run that gives the lowest value of the objective in Eqn. (16.3).

5. **K-Means++:** This is one of the simplest, reliable ways to initialize K-means. The goal of K-means++ is to choose the initial centers to be relatively far from one another: (1) Choose one datapoint at random to be the first center; (2) Compute the distance between each data point and its closest center, denoted $D(\mathbf{y}_i)$ for the ith datapoint; (3) Choose the next centre from the remaining datapoints with probability proportional to $D(\mathbf{y}_i)^2$ for the ith point. Once the new center is chosen, we steps (2) and (3) until K centers have been chosen.

Another key question is how one chooses the number of clusters, i.e., K. A common approach is to fix K based on some prior knowledge or computational constraints. One can also try different values of K, adding another term to to the objective function to penalize model complexity.

16.2 Hierarchical K-Means

One problem with k-means is that one has to store and search all K centers if, for example, one is given a query data point and asked to find the closest cluster. As the size of the training dataset grows, one may end up with enough clusters that this is a problem.

One way to improve search times is to use *hierarchical K-means*. Suppose that as a first stage of training one uses K-means but with only a small number of M clusters (e.g., 32). Then, for each of these M clusters one applies K-means again with another m clusters. Repeating this one more time we have a tree with depth 3, and M^3 clusters in the leaves of the tree. For a tree of depth L the tree would comprise $K = M^L$ clusters in total. Given a query, the (approximate) search process would simply traverse the tree, following the branch closest to the query at every level. This only provides approximate nearest neighbours. To reduce errors in such a greedy algorithm one can also choose to traverse a small number of clusters at each level of the tree.

16.3 Product Quantization

While hierarchical K-means reduces search times compared to exhaustive linear scan of all K centers, it is approximate and still requires that one store all K centers. A state of the art alternative is a well-known method for vector quantization called *Product Quantization*.

The idea is relatively simple, but highly effective. Suppose you break the input vectors in half, each of which has dimension $d/2$ (let's assume d is even for simpplicity). For an image this would mean breaking each image into 2 halves, perhaps by breaking each image into the left half and the right half. And suppose we used K-means with M clusters one the left halves, and then on the right halves. Each image of the input is then approximated by finding a nearest cluster for the left half, and one for the right. Given M clusters centers for the left and M for the right, one can combine them in M^2 possible ways. The squared error in the approximation will then be equal to the squared error for the approximation of the left half, plus the squared error in the approximation of the right half.

More generally, if one breaks the input data vectors into H parts, each of which is clustered with M clusters, then there are a total of $K = M^H$ ways to approximate any input vector. We therefore have $K = M^H$ cluster centers. But to store all the centers requires storage that is just $O(M\,H)$. In other words, the number of centers grows exponentially in the cost of storing the centers – Product Quantization is an extremely effective methods for approxmative coding and search with massive datasets.

16.4 K-Medoids

K-medoids clustering is a variant of K-means with the additional constraint that the cluster centers must be drawn from the data. The following algorithm, called Farthest First Traversal, or Hochbaum-Shmoys, is simple and effective:

Randomly select a data point $\mathbf{y}_i$ as the first cluster center: $\mathbf{c}_1 \leftarrow \mathbf{y}_i$
for $j = 2$ to K
    // Find the data point furthest from all existing centers:
    $i \leftarrow \arg\max_i \min_{k<j} ||\mathbf{y}_i - \mathbf{c}_k||^2$
    $\mathbf{c}_j \leftarrow \mathbf{y}_i$
end for
Label all remaining data points according to their nearest centers (as in k-means)

This algorithm provides a quality guarantee: it gives a clustering that is no worse than twice the error of the optimal clustering.

K-medoids clustering can also be improved by coordinate descent. The labeling step is the same as in K-means. However, the cluster updates must be done by brute-force search each time a new cluster center candidate is considered.

16.5 Gaussian Mixture Models

The Mixtures-of-Gaussians (MoG) model can be viewed as a generalization of K-means clustering. While K-means clustering works well for well-separated clusters that are more or less spherical, the MoG model can handle the wider class of oblong clusters, and it does an excellent job when clusters are overlapping. To this end, MoG algorithms compute a "soft," probabilistic assignment of points to clusters rather than strictly assigning each point to one cluster. This allows the algorithm to better handle overlapping clusters.

The MoG model is also probabilistic. As such it is also used as a way to learn a generative probabilistic prior model for data (as can PPCA for example). Fitting a Gaussian density to data is useful when the data are effectively unimodal. The MoG model can be used to describe multimodal distributions, and is hence much more powerful.

The MoG model comprises a linear combination of K Gaussian distributions, each with its own mean and covariance $\{(\mu_j, \mathbf{C}_j)\}_{j=1}^K$. Each Gaussian component also has an associated (prior) probability m_j, such that $\sum_j m_j = 1$. These probabilities, often called mixing probabilities, essentially represent the fraction of the data assigned to (or generated by) the different Gaussian components. As a shorthand, it is convenient to capture all model parameters with a single variable, i.e., $\theta = \{m_{1:K},\ \mu_{1:K},\ \mathbf{C}_{1:K}\}$. When used for clustering, the hope is that each Gaussian component in the mixture should correspond to a single cluster.

The complete generative model comprises the prior probability of each Gaussian component, and a Gaussian likelihood over the data (or feature) space for each component:

$$P(\ell = j \,|\, \theta) \;=\; m_j \tag{16.4}$$

$$p(\mathbf{y} \,|\, \theta, \ell = j) \;=\; G(\mathbf{y};\, \mu_j, \mathbf{C}_j) \tag{16.5}$$

where ℓ is the random variable that represents which of the K Gaussian components the data point was generated by.[1] To sample a single data point from this (generative) model, we first randomly select a Gaussian component (i.e., choose a value for ℓ), according to a categorical distribution with probabilities $\{m_j\}$. This gives us a value for ℓ between 1 and K. Then, we randomly sample a point from the selected Gaussian component. The likelihood of a single data point can be derived

[1] Much like the assignment variables we used above in the description of the K-means algorithm.

by the product rule and the sum rule:

$$p(\mathbf{y} \,|\, \theta) \;=\; \sum_{j=1}^{K} p(\mathbf{y}, \ell = j \,|\, \theta) \tag{16.6}$$

$$\;=\; \sum_{j=1}^{K} p(\mathbf{y} \,|\, \ell = j, \theta)\; P(\ell = j \,|\, \theta) \tag{16.7}$$

$$\;=\; \sum_{j=1}^{K} m_j \, \frac{1}{\sqrt{(2\pi)^d |C_j|}} e^{-\frac{1}{2}(\mathbf{y}-\mu_j)^T \mathbf{C}_j^{-1} (\mathbf{y}-\mu_j)} \tag{16.8}$$

where d is the dimension of training data vectors $\mathbf{y}$.

As mentioned at the outset, this model is formulated as a linear combination (or blend) of Gaussian distributions. Unlike PPCA which fits a single Gaussian density to the data, here we obtain a multi-modal distribution, comprising the weighted sum of unimodal Gaussians. Interestingly, the MoG model is similar to the Gaussian class-conditional model that we used for classification; the difference is that the class labels are not given to us here, and hence they need to be inferred.

In general, the approach of building models by mixtures is quite general and can be used for many other types of distributions as well. For example, we could build a mixture of Student-t distributions, or a mixture of a Gaussian and a uniform, and so on. But here we'll restrict our attention to Gaussians.

16.5.1 Learning

Given a data set $\mathbf{y}_{1:N}$, where each data point is assumed to be drawn independently from the model, we learn the model parameters, θ, by minimizing the negative log-likelihood of the data:

$$\mathcal{L}(\theta) \;=\; -\ln p(\mathbf{y}_{1:N} \,|\, \theta) \;=\; -\sum_i \ln p(\mathbf{y}_i \,|\, \theta) \tag{16.9}$$

Note that this is a constrained optimization, since we require $m_j \geq 0$ and $\sum_j m_j = 1$. Furthermore, $\mathbf{C}_j$ must be symmetric, positive-definite matrix to be a covariance matrix. Unfortunately, this optimization cannot be performed in closed-form.

One approach to optimization is to use gradient descent. But there are a few issues associated with doing so. First, some care is required to avoid numerical issues (discussed below). Second, this learning is a constrained optimization, due to constraints on the values of the m_js. One solution is to project onto the constraints during optimization; i.e., at each gradient descent step (and inside the line search loop), we simply set all negative a values to zero and renormalize the m_js so that they sum to one.

Another option is to reparameterize the problem to be unconstrained. Specifically, we could define new variables β_j, and define the m_js as functions of the β_js, e.g.,

$$m_j(\beta) = \frac{e^{\beta_j}}{\sum_{j=1}^{K} e^{\beta_j}} \tag{16.10}$$

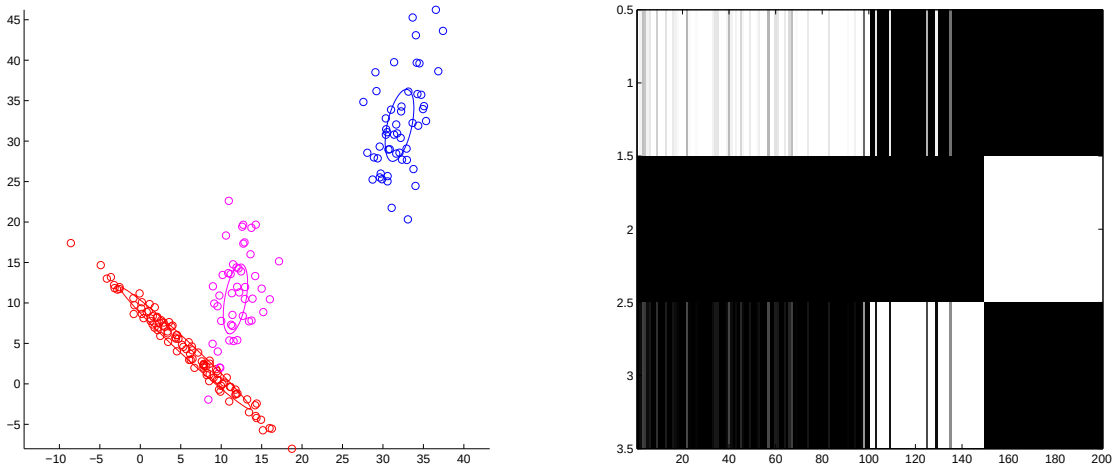

Figure 16.2: Mixture of Gaussians model applied to a dataset generated from three Gaussians. The resulting γ is visualized on the right. The data points are shown as colored circles. The color is determined by the cluster with the highest posterior assignment probability γ_{ij}. One standard deviation ellipses are shown for each Gaussian. Note that the blue points are well isolated and there is little ambiguity in their assignments. The other two distributions overlap, and one can see how the orientation and eccentricity of the covariance structure (the ellipses) influence the assignment probabilities.

This definition ensures that, for any choice of the β_js, the m_js will satisfy the constraints. We substitute this expression into the model definition and then optimize for the β_js instead of the m_js with gradient descent. Similarly, we can enforce the constraints on the covariance matrix by reparameterization. This is normally done using a upper-triangular matrix $\mathbf{U}$ such that $\mathbf{C} = \mathbf{U}^T\mathbf{U}$.

An alternative to gradient descent is the **Expectation-Maximization** algorithm (EM). EM is a quite general algorithm for "hidden variable" (or "missing data") problems. In this case the missing (or unobserved) data are the labels ℓ that specify the cluster to which data point is assigned (or generated by). In EM, we define a probabilistic labeling variable $\gamma_{i,j}$, often called the ownership probabilities. The variable $\gamma_{i,j}$ corresponds to the probability that data point i came from cluster j. In other words, $\gamma_{i,j}$ is meant to estimate $P(\ell = j|\mathbf{y}_i, \theta)$. In EM, we optimize both θ and γ together. The algorithm alternates between the *E-step*, which updates the γs, and the *M-step*, which updates the model parameters θ.

pick initial values for γ and θ
loop
  // E-step:
  for each data point i **do**
    $\gamma_{i,j} \leftarrow P(\ell = j \,|\, \mathbf{y}_i, \theta)$
  end for

// M-step:
for each cluster j **do**
$m_j \leftarrow (\sum_i \gamma_{i,j}) \,/\, N$
$\mu_j \leftarrow (\sum_i \gamma_{i,j} \mathbf{y}_i) \,/\, (\sum_i \gamma_{i,j})$
$\mathbf{C}_j \leftarrow (\sum_i \gamma_{i,j} (\mathbf{y}_i - \mu_j)(\mathbf{y}_i - \mu_j)^T) \,/\, (\sum_i \gamma_{i,j})$
end for
end loop

Note that the E-step is the same as classification in the Gaussian class-conditional model.

The EM algorithm is a local optimization algorithm, and so the results will depend on initialization. Initialization strategies similar to those used for K-means above can be used.

16.5.2 Numerical Issues

Exponentiating very small negative numbers can often lead to underflow when implemented in floating-point arithmetic, e.g., e^{-A} will give zero for large A, and $\ln e^{-A}$ will give an error (or `-Inf`) whereas it should return $-A$. These issues will often cause machine learning algorithms to fail. MoG has several steps which are susceptible. Fortunately, there are some simple tricks that can be used.

1. Many computations can be performed directly in the log domain. For example, it may be more stable to compute
$$ae^b \tag{16.11}$$
as
$$e^{\ln a + b} \tag{16.12}$$
This avoids issues where b is negative and so small that e^b evaluates to zero in floating point, but ae^b is much greater than zero.

2. When computing an expression of the form:
$$\frac{e^{-\beta_j}}{\sum_j e^{-\beta_j}} \tag{16.13}$$
large values of β could lead to the above expression being zero for all j, even though the expression must sum to one. This may arise, for example, when computing the γ updates, which have the above form. The solution is to make use of the identity:
$$\frac{e^{-\beta_j}}{\sum_j e^{-\beta_j}} = \frac{e^{-\beta_j + c}}{\sum_j e^{-\beta_j + c}} \tag{16.14}$$
for any value of c. A suitable choice to prevent underflow is $c = \min_j \beta_j$.

3. Underflow can also occur when evaluating
$$\ln \sum_i e^{-\beta_j} \tag{16.15}$$

which can be fixed by using the identity

$$\ln \sum_i e^{-\beta_j} \;=\; \left(\ln \sum_i e^{-\beta_j + c} \right) - c \tag{16.16}$$

16.5.3 Free Energy

There are several ways to derive this algorithm for fitting mixture models. One approach is to directly minimize the negative log likelihood with a constraint on the mixing probabilities (formulated as an unconstrained optimization with a Langrange multiplier to ensure the mixing probabilities sum to 1).

Another approach, which is somewhat more powerful and helps one understand the EM algorithm as used here and its variants, is based on a quantity called the **Free Energy**, denoted $\mathcal{F}$ and defined as

$$\mathcal{F}(\theta, \gamma) \;=\; -\sum_{i,j} \gamma_{i,j} \ln p(\mathbf{y}_i, \ell = j | \theta) \;+\; \sum_{i,j} \gamma_{i,j} \ln \gamma_{i,j} \;. \tag{16.17}$$

The first term is a sum over all data points of the expected negative log likelihood of the data and the cluster assignments (the missing data) under the posterior probability of the labels, i.e., $\sum_i E_{\ell \,|\, \mathbf{y}_i, \theta}[\ln p(\mathbf{y}_i, \ell \,|\, \theta)]$. The second term is a sum over the data points of the negative entropy of the label posterior probabilities, i.e., $\sum_i H(\ell \,|\, \mathbf{y}_i, \theta)$. In the specific context of the Gaussian mixture model, the free energy is given by

$$\begin{aligned} \mathcal{F}(\theta, \gamma) \;=\; & \frac{1}{2} \sum_{i,j} \gamma_{i,j} (\mathbf{y}_i - \mu_j)^T \mathbf{C}_j^{-1} (\mathbf{y}_i - \mu_j) \\ & + \frac{1}{2} \sum_{i,j} \gamma_{i,j} \ln (2\pi)^d |\mathbf{C}_j| \;-\; \sum_{i,j} \gamma_{i,j} \ln m_j \\ & + \sum_{i,j} \gamma_{i,j} \ln \gamma_{i,j} \end{aligned} \tag{16.18}$$

The EM algorithm is a coordinate descent algorithm for optimizing the free energy, subject to the constraint that $\sum_j \gamma_{i,j} = 1$ and the constraints on the mixing probabilities m_j. In other words, EM can be written compactly as:

pick initial values for γ and θ
loop
// E-step:
$\gamma \leftarrow \arg\min_\gamma \mathcal{F}(\theta, \gamma)$
// M-step:
$\theta \leftarrow \arg\min_\theta \mathcal{F}(\theta, \gamma)$
end loop

You will notice that the free energy is not identical to the negative log-likelihood $\mathcal{L}(\theta)$ that we initially set out to minimize. Fortunately, the free energy has the following important properties:

- When the value of γ is optimal, the free energy is equal to the negative log-likelihood (defined above in (16.9)):

$$\mathcal{L}(\theta) \;=\; \min_{\gamma} \mathcal{F}(\theta, \gamma) \tag{16.19}$$

  We can use this fact to evaluate the negative log-likelihood simply by running an E-step and then computing the free energy. In fact, this is often more efficient than directly computing the negative log-likelihood. The proof is given in the next section.

- The minima of the free energy are also minima of the negative log-likelihood:

$$\min_{\theta} \mathcal{L}(\theta) \;=\; \min_{\theta,\gamma} \mathcal{F}(\theta, \gamma) \tag{16.20}$$

  (This follows from the previous property.) The key consequence of this property is that **optimizing the free energy is the same as optimizing the negative log-likelihood.** This is interesting because the negative log likelihood (16.9) does not even require that we define the label assignment probabilities γ.

- The Free Energy is an upper-bound on the negative log-likelihood:

$$\mathcal{F}(\theta, \gamma) \geq \mathcal{L}(\theta) \tag{16.21}$$

  for all values of γ. This observation gives a sanity check for debugging the free energy computation.

The Free Energy also provides a very helpful tool for debugging. Any step of an implementation that increases the free energy must be incorrect. The term Free Energy arises from its original definition in statistical physics.

16.5.4 Proofs

This content of this section is not required material for this course and you may skip it. Here we outline proofs for the key features of the free energy.

EM updates. The steps of the EM algorithm may be derived by solving $\arg\min_{\gamma} \mathcal{F}(\theta, \gamma)$ and $\arg\min_{\theta} \mathcal{F}(\theta, \gamma)$. In most cases, the derivations generalize familiar ones, e.g., weighted least-squares. The parameters a and γ specify probabilities of a multinomial distribution, and optimization of them requires Lagrange multipliers or reparameterization. One may ignore the positivity constraint, as it turns out to be automatically satisfied. The details will be skipped here.

Equality after the E-step. The E-step computes the optimal value for γ:

$$\gamma^* \leftarrow \arg\min_{\gamma} \mathcal{F}(\theta, \gamma) \tag{16.22}$$

which is given by:

$$\gamma^*_{i,j} = P(\ell = j | \mathbf{y}_i, \theta) \tag{16.23}$$

Substituting this into the free energy gives:

$$\begin{aligned}
\mathcal{F}(\theta, \gamma^*) &= -\sum_{i,j} P(\ell = j \,|\, \mathbf{y}_i, \theta) \, \ln \frac{p(\mathbf{y}_i, \ell = j, \theta)}{P(\ell = j \,|\, \mathbf{y}_i, \theta)} \\
&= -\sum_{i,j} P(\ell = j \,|\, \mathbf{y}_i, \theta) \, \ln p(\mathbf{y}_i, \theta) \\
&= -\sum_{i} \left(\ln p(\mathbf{y}_i, \theta) \, \sum_j P(\ell = j \,|\, \mathbf{y}_i, \theta) \right) \\
&= -\sum_{i} \ln p(\mathbf{y}_i, \theta) \\
&= \mathcal{L}(\theta)
\end{aligned} \tag{16.24}$$

Hence,

$$\mathcal{L}(\theta) = \min_{\gamma} \mathcal{F}(\theta, \gamma) \tag{16.25}$$

Bound. An important building block in proving that $\mathcal{F}(\theta, \gamma) \geq \mathcal{L}(\theta)$ is **Jensen's Inequality**, which applies since $\ln$ is a "concave" function and $\sum_j b_j = 1$, $b_j \geq 0$.

$$\ln \sum_j b_j x_j \geq \sum_j b_j \ln x_j, \quad \text{or} \tag{16.26}$$

$$-\ln \sum_j b_j x_j \leq -\sum_j b_j \ln x_j \tag{16.27}$$

We will not prove this here.

We can then derive the bound as follows:

$$\begin{aligned}
\mathcal{L}(\theta) &= -\sum_i \ln \sum_j p(\mathbf{y}_i, \ell = j, \theta) \\
&= -\sum_i \ln \sum_j \frac{\gamma_{i,j}}{\gamma_{i,j}} \, p(\mathbf{y}_i, \ell = j, \theta) \\
&\leq -\sum_{i,j} \gamma_{i,j} \ln \frac{p(\mathbf{y}_i, \ell = j, \theta)}{\gamma_{i,j}} \\
&= \mathcal{F}(\theta, \gamma)
\end{aligned} \tag{16.28}$$

(This assumes $\gamma_{i,j}$ are non-negative and sum to one over j for a given i.)

16.5.5 Relation to k-Means

It should be clear that the K-means algorithm is very closely related to fitting Gaussian mixture models. In fact, EM reduces to K-means if we make the following restrictions on the model:

- The class probabilities are equal: $m_j = \frac{1}{K}$.
- The Gaussians are spherical with identical variances: $\mathbf{C}_j = \sigma^2 \mathbf{I}$ for all j.
- The Gaussian variances are infinitesimal, i.e., we consider the algorithm in the limit as $\sigma^2 \to 0$. This causes the optimal values for γ to be binary, since, if j is the nearest class, $\lim_{\sigma^2 \to 0} P(\ell = j | \mathbf{y}_i) = 1$.

With these modifications, the Free Energy becomes equivalent to the K-means objective function, up to constant values, and the EM algorithm becomes identical to K-means.

16.5.6 Degeneracy

There is a degeneracy in the MoG objective function. Suppose we center one Gaussian at one of the data points, so that $\mathbf{c}_j = \mathbf{y}_i$. The error for this data point will be zero, and by reducing the variance of this Gaussian, we can always increase the likelihood of the data. In the limit as this Gaussian's variance goes to zero, the data likelihood goes to infinity. Hence, some effort may be required to avoid this situation. This degeneracy can also be avoided by using a more Bayesian form of the algorithm, e.g., marginalizing out the cluster centers rather than estimating them.

16.6 Determining the number of clusters

Determining the value of K is a model selection problem: we want to determine the most-likely value of K given the data. Cross validation is not appropriate here, since we do not have any supervision (e.g., correct labels from a subset of the data). Bayesian model selection can be employed, e.g., by maximizing

$$K^* \;=\; \arg\max_K P(K|\mathbf{y}_{1:N}) \;=\; \arg\max_K \int p(K, \theta \,|\, \mathbf{y}_{1:N}) d\theta \tag{16.29}$$

where θ are the model parameters. This evaluation is somewhat mathematically-involved. A very coarse approximation to this computation is Bayesian Information Criterion (BIC).

