1 Gaussian Mixture Models

Lab Objective: *Understand the formulation of Gaussian Mixture Models (GMMs) and use the Expectation Maximization algorithm to estimate GMM parameters.*

Mixture models are a useful way to combine distributions together that allows us to describe much more complicated distributions than using just the standard list of named distributions. The essential idea of a mixture model is in its name: it is a mixture of several different models, or probability distributions. Each of these model is called a *component*. Each component has a certain probability associated with it, called its *weight*, that describes how likely it is for a sample from the model to come from that component. We denote the weight of the i-th component as w_i.

In this lab, we focus on *Gaussian Mixture Models*, or GMMs for short. In a GMM, each component is a multivariate Gaussian (normal) distribution. Each of these is parameterized by a mean μ_i and a covariance matrix Σ_i.

A GMM with K components thus has parameters $\theta = (w_1, \dots, w_K, \mu_1, \dots, \mu_K, \Sigma_1, \dots, \Sigma_K)$. We can use the law of total probability to evaluate the density of a GMM, which is given by

$$P(z|\theta) = \sum_{k=1}^{K} w_k \mathcal{N}(z|\mu_k, \Sigma_k)$$

where

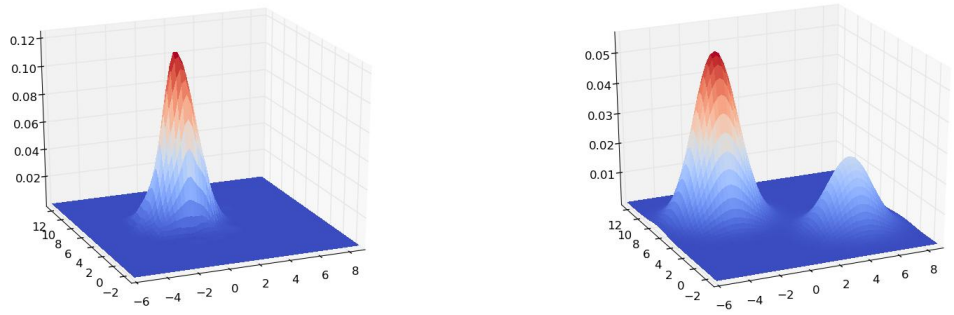
$$\mathcal{N}(z|\mu, \Sigma) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\left(-\frac{1}{2}(z-\mu)^T \Sigma^{-1} (z-\mu)\right)$$

is the density function of a multivariate normal distribution.

It is important to keep in mind that a GMM does *not* arise from adding weighted multivariate normal random variables, but rather from weighting the responsibility of each multivariate normal random variable. The first case simply results in a different multivariate normal distribution. Refer to Figure 1.1 for a visualization of these two cases.

Problem 1. Throughout this lab, we will build a GMM class with various methods. Write the `__init__` method for this class. It should accept a parameter for the number of components and optional parameters for the weights, means, and covariance matrices which define the GMM, and store these.[a]

If we have K components and d dimensions, then the weights should have shape `(K,)`, the means `(K,d)`, and the covariances `(K,d,d)`. The parameters for the k-th component can



(a) Sum of weighted multivariate normal random variables.

(b) Weighted mixture of multivariate normal random variables.

Figure 1.1

be found as `weights[k]`, `means[k]`, `covars[k]`.

[a] If we don't have a good guess for the parameters of the GMM to pass into the class, it makes more sense to initialize these from the dataset we are training on, which we will do later in the `fit` method; hence, we let the parameters be optional here.

Problem 2. Write a method `component_logpdf` for your class that accepts a component `k` and a point `z` and computes

$$\log w_k + \log \mathcal{N}(z|\mu_k, \Sigma_k),$$

the logarithm of the contribution of the k-th component of the pdf. Also write a method `pdf` that accepts a point `z` and returns the probability density of the whole GMM at that point.

Hint: `scipy.stats.multivariate_normal.pdf` and `scipy.stats.multivariate_normal.logpdf` can be used to efficiently evaluate the multivariate normal pdf.

To test your functions, create the following GMM:

```
gmm = GMM(n_components = 2,
        weights = np.array([0.6, 0.4]),
        means = np.array([[-0.5, -4.0], [0.5, 0.5]]),
        covars = np.array([
            [[1, 0],[0, 1]],
            [[0.25, -1],[-1, 8]],
        ]))
```

Your functions should give the following output:

```
>>> gmm.pdf(np.array([1.0, -3.5]))
0.05077912539363083
```

```
# Component 0
>>> gmm.component_logpdf(0, np.array([1.0, -3.5]))
-3.598702690175336
# Component 1
>>> gmm.component_logpdf(1, np.array([1.0, -3.5]))
-3.7541677982835004
```

Note that since this GMM is 2-dimensional, the input point must be an array of length 2.

In order to draw a value from a mixture model, we must first draw a variable $X \sim \text{Cat}(w_1, \ldots, w_K)$ that represents which component the sample comes from. We can then draw the sample $Z \sim \mathcal{N}(\mu_X, \Sigma_X)$. If we want to draw multiple samples, we need to repeat this process for each one (draw an X and then draw a Z.

Problem 3. Write a method `draw` for the GMM class that randomly draws from the model. If `m` points are drawn and the GMM is d-dimensional, the returned array should have shape `(m,d)`.

Draw a sample of 10,000 points from the GMM defined in Problem 2. Plot the pdf of the GMM (using `plt.pcolormesh`) and a hexbin plot of the drawn points. How do the plots compare?

The following code can be used to plot the pdf:

```
## Create the grid to plot on
x = np.linspace(-8,8,100)
y = np.linspace(-8,8,100)
X, Y = np.meshgrid(x, y)
## Calculate the pdf at each point
# If your pdf function uses array broadcasting, you can do the following:
Z = gmm.pdf(np.dstack((X,Y)))
# Otherwise, you need to iterate over each point:
Z = np.array([[
        gmm.pdf([X[i,j], Y[i,j]]) for j in range(100)
        ] for i in range(100)
    ])
## Create the plot
plt.pcolormesh(X, Y, Z, shading='auto')
```

We now consider how to estimate the parameters of a GMM given some observed data $Z = z_1, \ldots, z_n$. Ordinarily, a good approach would be to try to directly maximize the log-likelihood

$$l(\theta) = \sum_{i=1}^{n} \log \sum_{j=1}^{K} w_j \mathcal{N}(z_i | \mu_j, \Sigma_j).$$

However, this expression is very difficult to deal with using standard optimization methods, particularly because of the sum inside of the logarithm. A good alternative in this case is the *expectation*

maximization (EM) algorithm. This is an iterative algorithm, where each step consists of maximizing a function that is designed to approximate the log-likelihood while being much easier to maximize.

Each iteration consists of two steps, the E-step and the M-step. Suppose our estimated parameters at the t-th iteration are $\theta^t = (w_1^t, \ldots, w_K^t, \mu_1^t, \ldots, \mu_K^t, \Sigma_1^t, \ldots, \Sigma_K^t)$. Note that t is an index, not an exponent. For each data point $z_i, 1 \leq i \leq n$ and each component $1 \leq k \leq K$, the E-step consists of computing

$$\begin{aligned} q_i^t(k) &= P(X_i = k | z_i, \theta^t) \\ &= \frac{P(z_i | X_i = k, \theta^t)}{P(z_i | \theta^t)} \\ &= \frac{w_k^t \, \mathcal{N}(z_i | \mu_k^t, \Sigma_k^t)}{\sum_{k'=1}^K w_{k'}^t \, \mathcal{N}(z_i | \mu_{k'}^t, \Sigma_{k'}^t)} \end{aligned}$$

In order to accurately compute this quantity, however, we need to be more careful. It is possible that due to floating point underflow[1] that each term $w_{k'}^t \, \mathcal{N}(z_i | \mu_{k'}, \Sigma_{k'})$ in the sum in the denominator becomes zero, which is a major problem. This particularly happens if the exponents in the multivariate normal densities all are large negative numbers. To avoid this problem, we can rescale the numerator and denominator. Let

$$\ell_{i,k} = \log w_k^t + \log \mathcal{N}(z_i | \mu_k^t, \Sigma_k^t),$$

the logarithm of each term in the denominator. For each data point z_i, we can find

$$L_i = \max_{k'} \ell_{i,k'},$$

the largest of these logarithms. Then, we can rewrite the quantity we want to calculate as

$$\begin{aligned} q_i^t(k) &= \frac{w_k^t \, \mathcal{N}(z_i | \mu_k^t, \Sigma_k^t)}{\sum_{k'=1}^K w_{k'}^t \, \mathcal{N}(z_i | \mu_{k'}^t, \Sigma_{k'}^t)} \\ &= \frac{e^{\ell_{i,k}}}{\sum_{k'=1}^K e^{\ell_{i,k'}}} \\ &= \frac{e^{\ell_{i,k}} e^{-L_i}}{\sum_{k'=1}^K e^{\ell_{i,k'}} e^{-L_i}} \\ &= \frac{e^{\ell_{i,k} - L_i}}{\sum_{k'=1}^K e^{\ell_{i,k'} - L_i}}. \end{aligned}$$

This rescaling makes the largest term in the denominator equal to 1, so computing $q_i^t(k)$ in this way avoids underflow problems. Note that for the computation of any individual $q_i^t(k)$, the value L_i is a scalar that is the same for all components; however, you will have as many of these values as you have data points. As a reminder, i corresponds to the index of a data point and k corresponds to which componet we are comparing it to.

[1] As a refresher, one way that floating point numbers are limited is that they cannot represent positive numbers arbitrarily close to zero; at some point, if the number in a computation becomes too small, the computer is forced to round it to zero, which is called *underflow*. The threshhold is about 10^{-323} for the 64-bit floating point numbers used in python. Even if underflow does not occur, very small floating points have greatly reduced precision, so it is generally good to avoid using them.

Problem 4. Write a method `_compute_e_step` that calculates the $q_i^t(k)$ as given by the E-step, given a collection of observations. Be sure to do the calculation in a way that avoids underflow, and use array broadcasting when possible.

Your method will accept an array of shape `(n,d)`, where `n` is the number of data points and `d` is the dimensionality of the data (i.e. each row is a data point). The array you produce should have shape `(n_components, n)` where `result[k,i]` $= q_i^t(k)$ (i.e. each row is one component, and each column is a data point). The various intermediate values should have shapes similar to the following:

- The array of $\ell_{i,k}$s should have shape `(n_components, n)`
- The array of L_is should have shape `(n,)`
- The array of the denominator values $\sum_{k'=1}^{K} e^{\ell_{i,k'} - L_i}$ should also have shape `(n,)`

With the GMM from the example in Problem 2, you should get the following results:

```
>>> data = np.array([
        [0.5, 1.0],
        [1.0, 0.5],
        [-2.0, 0.7]
    ])
>>> gmm._compute_e_step(data)
array([[3.49810771e-06, 5.30334386e-05, 9.99997070e-01],
       [9.99996502e-01, 9.99946967e-01, 2.93011749e-06]])
```

Now that we have the $q_i^t(k)$, we can perform the M-step. This step consists of maximizing the function

$$Q^t(\theta) = \sum_{i=1}^{n} \sum_{k=1}^{K} q_i^t(k) \log w_k^t \mathcal{N}(z_i | \mu_k, \Sigma_k)$$

We then set

$$\theta^{t+1} = \underset{\theta}{\operatorname{argmax}} Q^t(\theta)$$

and iterate until the method appears to converge. In the case of GMMs, the maximizer θ^{t+1} of $Q^t(\theta)$ is given by

$$w_k^{t+1} = \frac{1}{n} \sum_{i=1}^{n} q_i^t(k)$$

$$\mu_k^{t+1} = \frac{\sum_{i=1}^{n} q_i^t(k) z_i}{\sum_{i=1}^{n} q_i^t(k)}$$

$$\Sigma_k^{t+1} = \frac{\sum_{i=1}^{n} q_i^t(k)(z_i - \mu_k^{t+1})(z_i - \mu_k^{t+1})^\top}{\sum_{i=1}^{n} q_i^t(k)}$$

For details on the derivation of the maximizer, refer to the Volume 3 textbook.

Problem 5. Write a method `_compute_m_step` for your GMM class that performs a single iteration of the EM algorithm. Use your function from Problem 4 to calculate the $q_i^t(k)$ values. Return the updated parameters (weights, means, and covariance matrices), as given by the M-step. Be sure to use array broadcasting when possible. The function `np.einsum` may be helpful for calculating Σ_k^{t+1}.

With the same GMM and data as in Problem 4, you should get the following results:

```
>>> gmm._compute_m_step(data)
(array([0.3333512, 0.6666488]),
 array([[-1.99983216,  0.69999044],
        [ 0.74998978,  0.75000612]]),
 array([[[ 4.99109197e-04, -2.91933135e-05],
         [-2.91933135e-05,  2.43594533e-06]],

        [[ 6.25109881e-02, -6.24997069e-02],
         [-6.24997069e-02,  6.24999121e-02]]]))
```

Problem 6. Write a `fit` method for your GMM class.

First, if the GMM's parameters are uninitialized (set to `None`), initialize the parameters of the components. We want to do this in a way that the algorithm starts with reasonable values for the dataset. A good way to initialize the means is to randomly select points from the dataset. The covariance matrices can be initialized as diagonal matrices based on the variance of the data. Ensure that the weights you choose add up to 1.

Then, perform the expectation maximization algorithm. Use the functions you created in Problems 4 and 5 to calculate the parameters at each step. Repeat until the parameters converge. Use the following to measure the change in the parameters with each iteration:

```
change = (np.max(np.abs(new_weights - old_weights))
        + np.max(np.abs(new_means - old_means))
        + np.max(np.abs(new_covars - old_covars)))
```

Problem 7. The file `problem7.npy` contains a collection of data drawn from a two-dimensional GMM. Train a GMM on this data with `n_components=3`. Plot the pdf of your trained GMM (in the same way as in Problem 3), as well as a hexbin plot of the training data. Your class should take less than 15 seconds to train on this dataset.

Hint: if you plot the pdf on too wide of a range, it may just look like a blob. For this one, a range of $[-4, 4] \times [-4, 4]$ works well.

Clustering with GMMs

An important use of mixture models is for *clustering*. The objective of clustering is to take an unlabeled dataset and separate it into some number of clusters, which can then be labeled. This is an instance of *unsupervised learning*, as it is a machine learning task where the the training algorithm does not need the true answers (in this case, the actual clusters).

In order to cluster a dataset using a GMM, we first need to train the GMM on that data. Then, we can assign each point a label by finding which component has the largest contribution to the pdf there. Written symbolically, for a data point z, we have

$$\text{Cluster}(z) = \text{argmax}_k w_k \mathcal{N}(z|\mu_k, \Sigma_k).$$

Note that the number of clusters (components) is a hyperparameter that must be selected before a GMM is trained. In general, cross-validation or some other method must be used to find the right number of clusters.

Problem 8. Write a `predict` method for your class. Given a set of data points, return which cluster has the highest pdf density for each data point.

The file `classification.npz` contains a set of 3-dimensional data points (`X`) and their labels (`y`). Use your class with `n_components=4` to cluster the data. Plot the points with the predicted and actual labels, and compute and return your model's accuracy. Your class should take less than 30 seconds to train on this dataset.

Note that the labels may be permuted; for instance, your model might cluster the points correctly, but swap the labels of clusters 1 and 2 compared to the true labels. The model would still be considered accurate in this case; we only care what the clusters are, not how the model labels them. To resolve this problem, we need to find the permutation of the labels that results in the highest accuracy. The following function does this in a way that is more efficient than directly checking all permutations:

```
from scipy.optimize import linear_sum_assignment
from sklearn.metrics import confusion_matrix

def get_accuracy(pred_y, true_y):
    """
    Helper function to calculate the actually clustering accuracy,
    accounting for the possibility that labels are permuted.
    """
    # Compute confusion matrix
    cm = confusion_matrix(pred_y, true_y)
    # Find the arrangement that maximizes the score
    r_ind, c_ind = linear_sum_assignment(cm, maximize=True)
    return np.sum(cm[r_ind, c_ind]) / np.sum(cm)
```

For convenience, a method `fit_predict` for the class is also included in the specifications file that calls both `fit` and `predict` to make the clustering process simpler.

Clustering with GMMs is closely related to the K-means algorithm. In fact, K-means can be viewed as a special case of GMMs. We now compare the effectiveness of GMMs for classification on this dataset with K-means, as well as comparing to sklearn's implementation.

Problem 9. Again using `classification.npz`, compare your class, sklearn's GMM implementation, and sklearn's K-means implementation for speed of training and for accuracy of the resulting clusters. Print your results. Be sure to check for permuted labels.

You should find that sklearn's GMM is actually faster on this dataset than K-means. This is in part because the dataset is rather low-dimensional. As the dimension of the dataset grows, GMMs suffer computationally from the curse of dimensionality much more than the K-means algorithm.