07 - Unsupervised Learning - Clustering

October 26, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 7 Unsupervised Learning - Clustering

1.1.1 Readings:

- [VANDER] Ch5
- [HASTIE] Ch14.3
- https://scikit-learn.org/stable/modules/clustering.html

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.2 K-Means Clustering

In the clustering problem, we are given a training set $\{x^{(1)}, \cdots, x^{(m)}\}$, and want to group the data into a few cohesive "clusters". Clustering algorithms seek to learn, from the properties of the data, an optimal division or discrete labeling of groups of points. Here, $x^{(i)} \in \mathbb{R}^n$ as usual, but **no labels $y^{(i)}$ are given**. Thus, this is an unsupervised learning problem.

The k-means clustering algorithm is as follows:

1. Define k
2. Initialize cluster centroids $\mu_1, \mu_2, \cdots, \mu_k \in \mathbb{R}^n$ randomly
3. Repeat until convergence: {

   For every $i \in m$, set

$$c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2$$

   where the right part is simply the Euclidean distance equation

$$d(x^{(i)}, x^{(i')}) = \sum_{j=1}^{n} (x^{ij} - x^{i'j})^2 = ||x^{(i)} - x^{(i')}||^2$$

   For each $j = 1, 2, \cdots, k$, set

$$\mu_j := \frac{\sum_{i=1}^{m} I\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} I\{c^{(i)} = j\}}$$

}

In the algorithm above, k is the number of clusters we want to find; and the cluster centroids μ_j represent our current guesses for the positions of the centers of the clusters. To initialize the cluster centroids, we could choose k training examples randomly, and set the cluster centroids to be equal to the values of these k examples. (Note that other initialization methods are also possible).

The inner loop of the algorithm repeatedly carries out two steps:

1. "Assigning" each training example $x^{(i)}$ to the closest cluster centroid μ_j, where *closest* is defined using Euclidean distance.
2. Moving each cluster centroid μ_j to the mean of the points assigned to it. The kth cluster centroid is the vector of the n feature means for the observations in the kth cluster

Choosing k

To choose k, we define "good" clusters as having minimum **within-cluster variation** $W(c_k)$, using the following formula:

$$W(c_k) = \sum_{i \in c_k} \sum_{j=1}^{n} (x^{ij} - \bar{x}^{kj})^2$$

Here $\bar{x}^{(k)}$ refers to the mean belong to class k. In other words, the within-cluster variation for the kth cluster is the sum of squared Euclidean distances between the observations and mean across all features in the kth cluster. Using this equation, we can come up with **total within-cluster variation** or sometimes we call **total sum of squares**

$$W(C) = \sum_{k=1}^{K} \sum_{i \in c_k} \sum_{j=1}^{n} (x^{ij} - \bar{x}^{kj})^2$$

We can say that best k should minimize the following objective function:

$$\min_k W(C)$$

Convergence

Is the k-means algorithm guranteed to converge? Yes it is, in a certain sense. In particular, let us define the **distortion function** to be

$$J(c, \mu) = \sum_{i=1}^{m} ||(x^{(i)} - \mu_{c^{(i)}}||^2$$

Thus, J measures the sum of squared distances between each training example $x^{(i)}$ and the cluster centroid $\mu_{c^{(i)}}$ to which it has been assigned. It can be shown that k-means is exactly coordinate descent on J. Specifically, the inner-loop of k-means repeatedly minimizes J with respect to c while holding μ fixed, while holding c fixed. Thus, J must monotonically decrease, and the value of J must converge.

The distortion function J is a non-convex function, and so coordinate desent on J is not guaranteed to converge to the global minimum. One common way to tackle this is to run k-means many times using different random initial values for the cluster centroids μ_j. Then, out of all the different clusterings found, pick the one that gives the lowest distortion $J(c, \mu)$

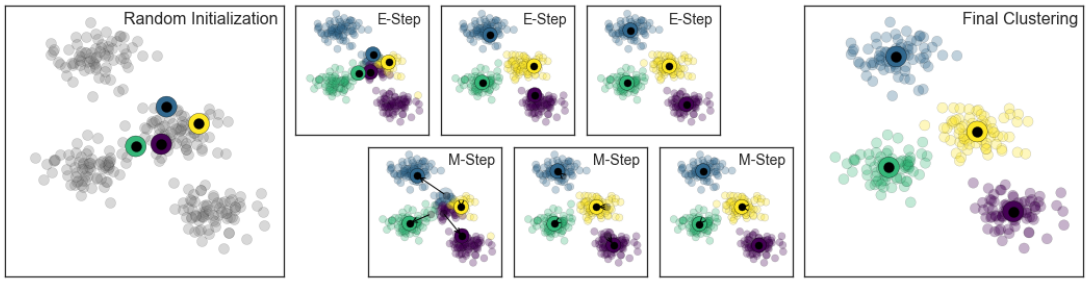
Relation to a bigger family of algorithm - Expectation-Maximization

Expectation–maximization (E–M) is a powerful algorithm that comes up in a variety of contexts within data science. k-means is a particularly simple and easy-to-understand application of the algorithm, and we will walk through it briefly here. In short, the expectation–maximization approach here consists of the following procedure:

1. Guess some cluster centers
2. Repeat until converged
    1. *E-Step*: assign points to the nearest cluster center
    2. *M-Step*: set the cluster centers to the mean

Here the "E-step" or "Expectation step" is so-named because it involves updating our expectation of which cluster each point belongs to. The "M-step" or "Maximization step" is so-named because it involves maximizing some fitness function that defines the location of the cluster centers—in this case, that maximization is accomplished by taking a simple mean of the data in each cluster. Each repetition of the E-step and M-step will always result in a better estimate of the cluster characteristics.

We can visualize the algorithm as shown in the following figure.



We shall soon explore a better EM algorithm called **Gaussian Mixture**.

1.2.1 Scratch

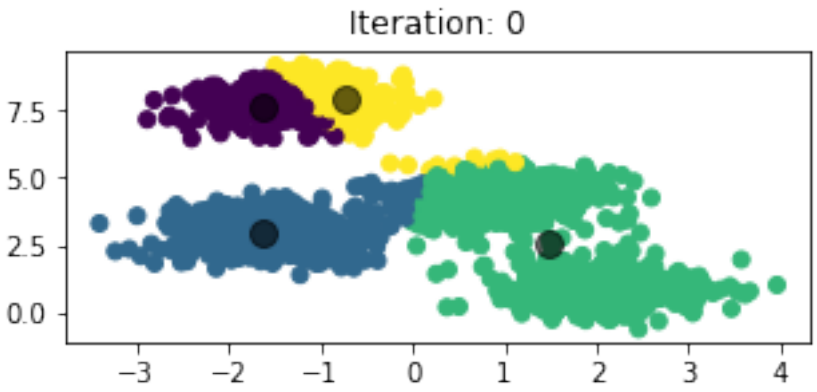
[2]:
```
#Implement K-means from scratch
from sklearn.datasets import make_blobs
from sklearn.metrics import pairwise_distances_argmin
from time import time

X, y_true = make_blobs(n_samples=1500, centers=4,
```

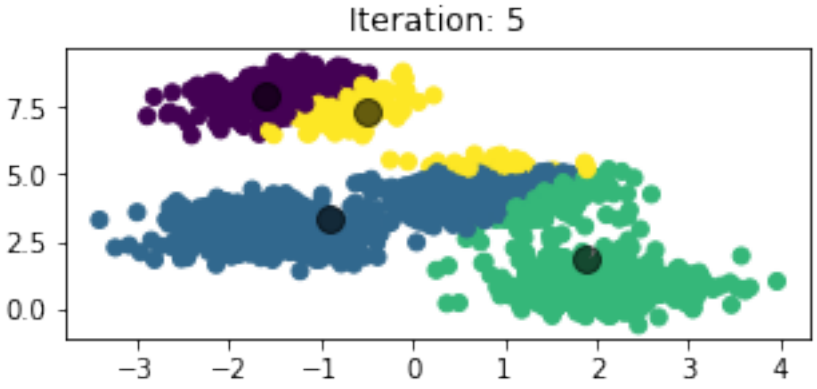
```
                        cluster_std=0.60, random_state=0)

def kmeans(X, n_clusters):
    m, n = X.shape

    #1. randomly choose n clusters from X
    #you can also randomly generate any two points
    rng = np.random.RandomState(42)
    i = rng.permutation(m)[:n_clusters]
    centers = X[i]

    iteration = 0

    while True:
        #2. assign lables based on closest center
        #return the index of centers having smallest
        #distance with X
        labels = pairwise_distances_argmin(X, centers)

        #3. find new centers
        new_centers = []
        for i in range(n_clusters):
            new_centers.append(X[labels == i].mean(axis=0))

        #convert list to np.array; you can actually combine #3
        #with np.array in one sentence
        new_centers = np.array(new_centers)

        #plotting purpose
        #plot every 5th iteration to save space
        #remove this if, if you want to see each snapshot
        if (iteration % 5 == 0):
            pred = pairwise_distances_argmin(X, new_centers)
            plt.figure(figsize=(5, 2))
            plt.title(f"Iteration: {iteration}")
            plt.scatter(X[:, 0], X[:, 1], c=pred)
            plt.scatter(new_centers[:, 0], new_centers[:, 1], s=100, c="black",␣
 ↪alpha=0.6)

        #4 stopping criteria - if centers do not
        #change anymore, we stop!
        if(np.allclose(centers, new_centers)):
            break
        else:
            centers = new_centers
            iteration+=1
```

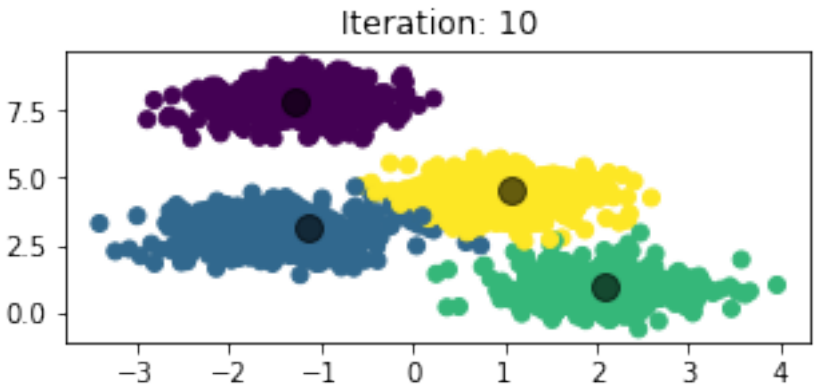
```
    print(f"Done in {iteration} iterations")
    return centers

def predict(X, centers):
    return pairwise_distances_argmin(X, centers)

start = time()
preds = predict(X, kmeans(X, n_clusters=4))
print(f"Fit and predict time: {time() - start}")
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=preds, s=50)
plt.title("Final result")
```

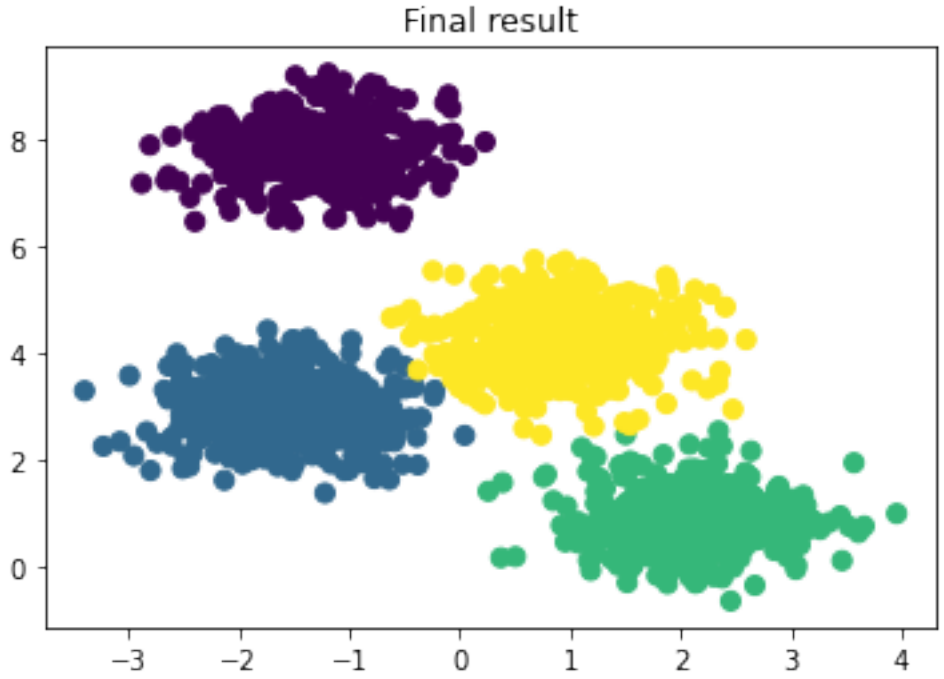
```
Done in 14 iterations
Fit and predict time: 0.061280012130737305
```

[2]: `Text(0.5, 1.0, 'Final result')`



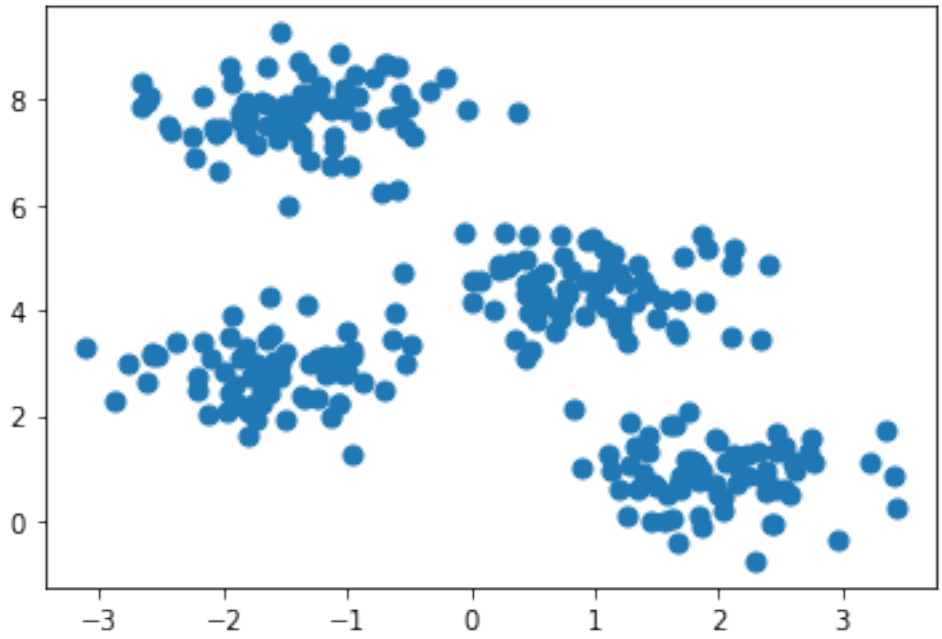






1.2.2 Sklearn

[3]:
```
from sklearn.datasets import make_blobs
import numpy as np
import matplotlib.pyplot as plt

X, y_true = make_blobs(n_samples=300, centers=4,
                       cluster_std=0.60, random_state=0)
plt.scatter(X[:, 0], X[:, 1], s=50);
```



K-mean is an unsupervised algorithm that aims to cluster the data based on distances.

[4]:
```
from sklearn.cluster import KMeans

#How did I know there are n_clusters = 4 (because I cheat!  But what if I don't␣
 ↪know in advance?)
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
print(y_kmeans)
```

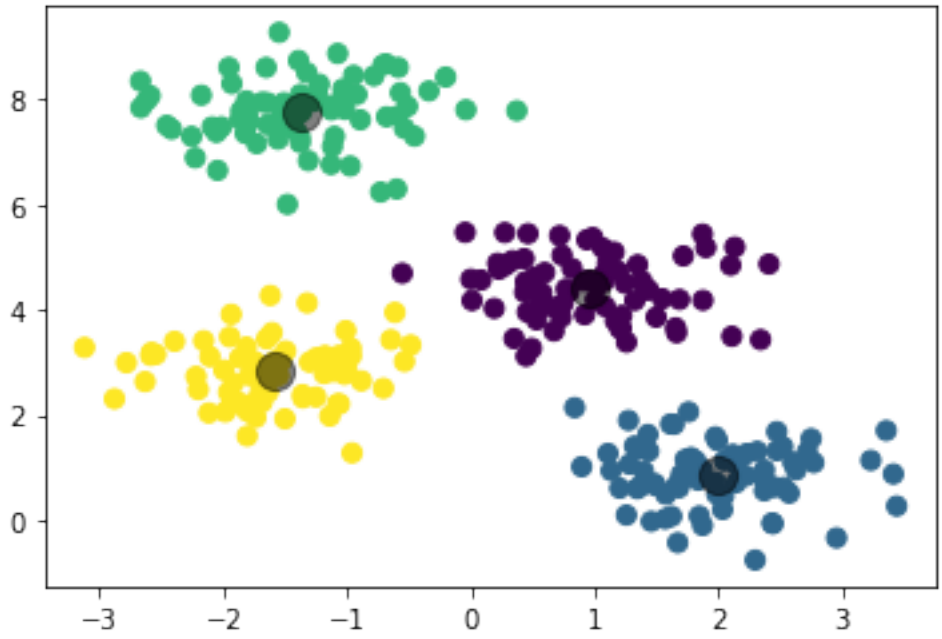
```
[1 2 0 2 1 1 3 0 2 2 3 2 0 2 1 0 0 1 3 3 1 1 0 3 3 0 1 0 3 0 2 2 0 2 2 2 2
 2 3 1 0 3 0 0 3 3 2 3 2 1 3 1 2 1 1 3 2 3 2 1 2 0 2 3 3 3 2 1 2 3 0 3 2 3
 3 2 3 0 1 2 1 0 1 1 2 0 1 0 2 2 0 1 2 3 3 0 1 1 0 3 2 1 2 1 0 1 1 0 2 0 3
 3 1 2 1 0 2 1 1 0 3 1 3 1 1 1 1 3 1 3 2 3 3 1 2 3 3 2 0 2 2 3 0 3 0 3 2 0
 2 2 2 0 2 0 1 3 2 3 1 0 2 0 0 1 0 3 3 0 1 0 0 2 1 0 3 2 1 1 0 3 1 0 3 3 0
 0 0 0 1 2 0 3 0 0 3 3 3 0 3 2 0 3 1 3 0 2 3 2 0 2 0 3 0 0 2 3 3 1 1 0 2 1
 1 3 1 3 0 2 2 0 0 2 0 1 3 0 1 3 2 3 1 0 1 2 2 2 2 3 3 2 0 3 1 0 3 3 3 1 1
 2 0 0 3 1 2 3 0 2 0 1 1 3 3 0 1 1 1 0 2 2 1 1 0 1 1 1 2 3 2 0 1 1 2 2 2 1
 1 0 2 3]
```

Let's visualize.

[5]:
```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

centers = kmeans.cluster_centers_
print("Centers: ", centers)
```

```
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
```

```
Centers:  [[ 0.94973532  4.41906906]
 [ 1.98258281  0.86771314]
 [-1.37324398  7.75368871]
 [-1.58438467  2.83081263]]
```

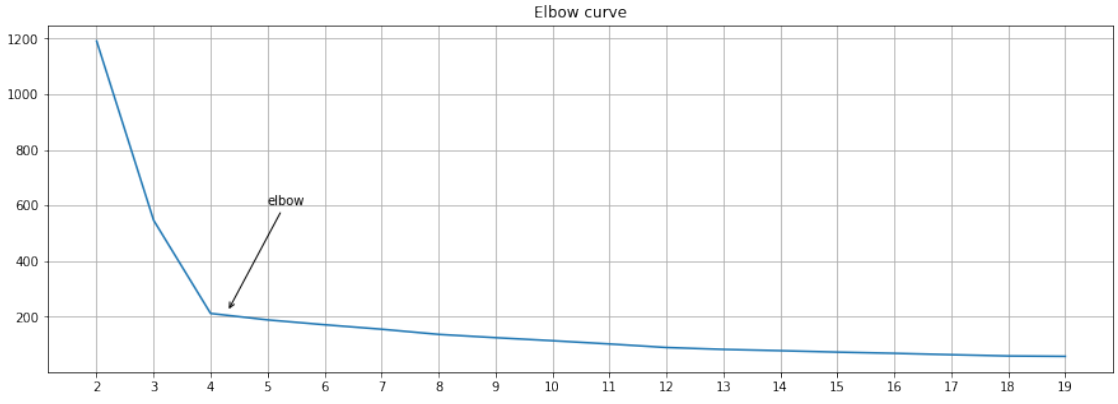


How to know how many clusters?

[6]:
```
#sum of squared distances
ssd = []
for k in range(2, 20):
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(X)
    ssd.append(kmeans.inertia_)

fig = plt.figure(figsize=(15, 5))
plt.plot(range(2, 20), ssd)
plt.xticks(range(2, 20))
plt.grid(True)
plt.title('Elbow curve')

plt.annotate('elbow', xy=(4.3, 220), xytext=(5, 600),  #xytext ---> xy
             arrowprops=dict(arrowstyle="->"))

'''
```

```
4 Clusters could be good. Why not 19 then?  Because 4 got a good balance␣
 ↪between within-sum of squared distances,
and the number of clusters.  Having way too many clusters with little gain of␣
 ↪the within-sum will increase
computation time.
'''
```

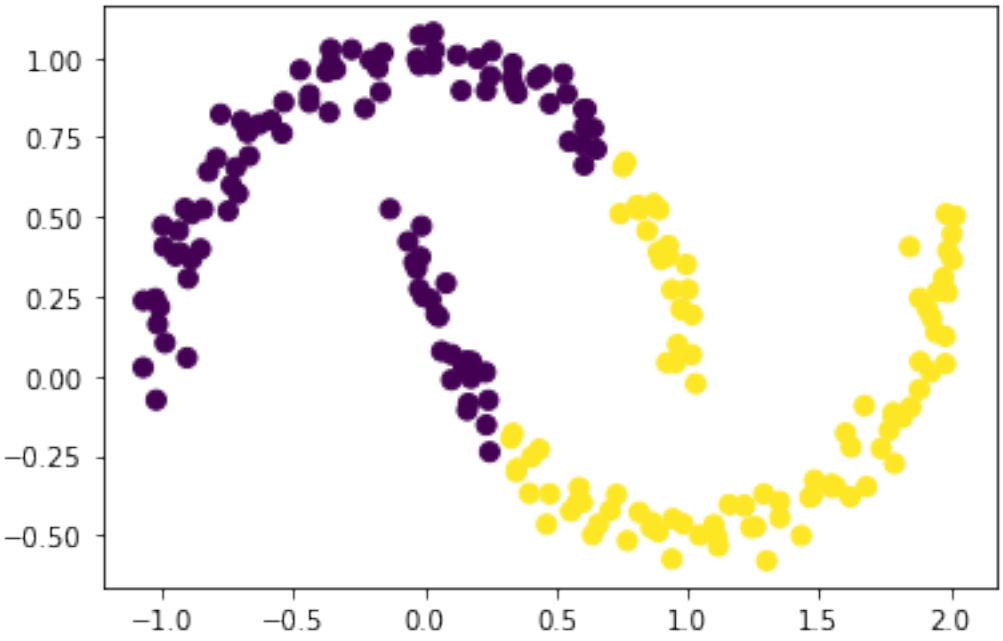
[6]:
```
'\n4 Clusters could be good. Why not 19 then?  Because 4 got a good balance
between within-sum of squared distances,\nand the number of clusters.  Having
way too many clusters with little gain of the within-sum will
increase\ncomputation time.\n'
```



K-means assumes equal-sized spherical distribution

[7]:
```
from sklearn.datasets import make_moons
X, y = make_moons(200, noise=.05, random_state=0)
```

[8]:
```
labels = KMeans(2, random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```



Example: k-means on digits Here we will attempt to use k-means to try to identify similar digits *without using the original label information*; this might be similar to a first step in extracting meaning from a new dataset about which you don't have any *a priori* label information.

We will start by loading the digits and then finding the `KMeans` clusters. Recall that the digits consist of 1,797 samples with 64 features, where each of the 64 features is the brightness of one pixel in an 8×8 image:

[9]:
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

[9]:
```
(1797, 64)
```

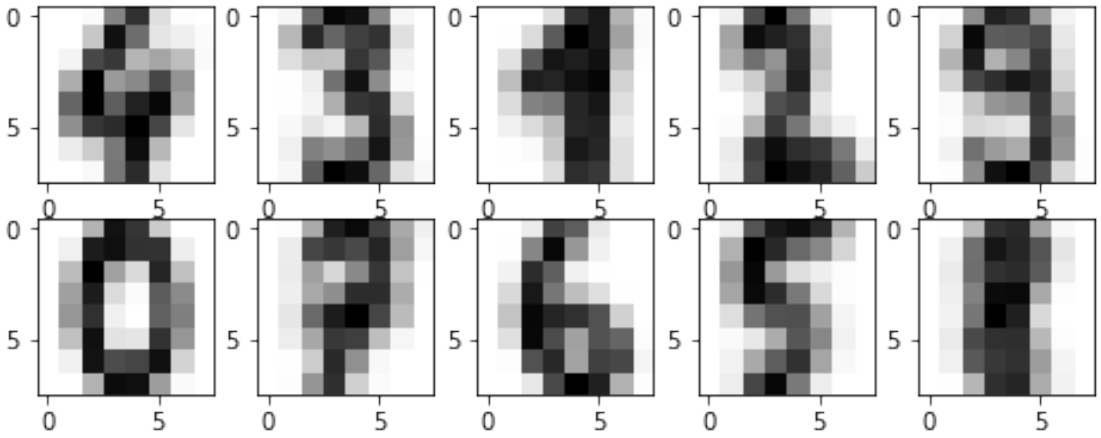
[10]:
```
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits.data)
kmeans.cluster_centers_.shape
```

[10]:
```
(10, 64)
```

The result is 10 clusters in 64 dimensions. Notice that the cluster centers themselves are 64-dimensional points, and can themselves be interpreted as the "typical" digit within the cluster. Let's see what these cluster centers look like:

[11]:
```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = kmeans.cluster_centers_.reshape(10, 8, 8)
```

```
for axi, center in zip(ax.flat, centers):
    axi.imshow(center, cmap=plt.cm.binary)
```



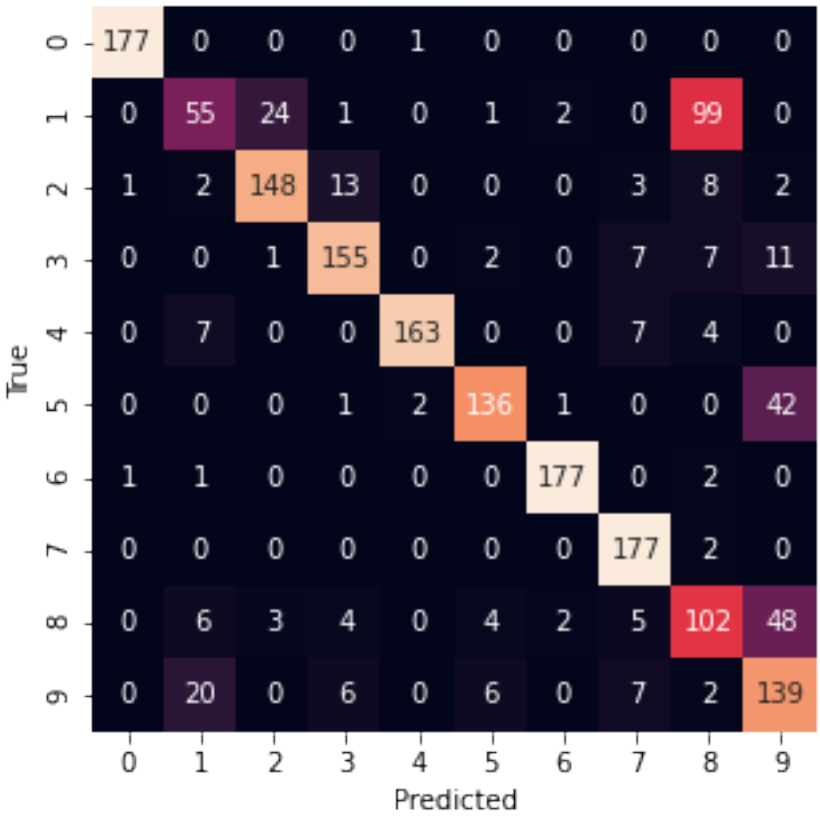
We see that *even without the labels*, `KMeans` is able to find clusters whose centers are recognizable digits, with perhaps the exception of 8.

Because k-means knows nothing about the identity of the cluster, the 0–9 labels may be permuted. We can fix this by matching each learned cluster label with the true labels found in them:

[12]:
```
from scipy.stats import mode
from sklearn.metrics import confusion_matrix
import seaborn as sns

pred = np.zeros_like(clusters)
for i in range(10):
    mask = (clusters == i)
    #mode to get the most correctly classified
    pred[mask] = mode(digits.target[mask])[0]

from sklearn.metrics import accuracy_score
print("Accuracy score: ", accuracy_score(digits.target, pred))
mat = confusion_matrix(digits.target, pred)
plt.figure(figsize=(16, 5))
sns.heatmap(mat, square=True, annot=True, fmt='d', cbar=False)
plt.xlabel("Predicted")
plt.ylabel("True")
```

```
Accuracy score:  0.7952142459654981
```

[12]:
```
Text(433.5, 0.5, 'True')
```



The main point of confusion is between the eights and ones. But this still shows that using k-means, we can essentially build a digit classifier *without reference to any known labels*!

Just for fun, let's try to push this even farther. We can use the t-distributed stochastic neighbor embedding (t-SNE) algorithm to pre-process the data before performing k-means.

[13]:
```
from sklearn.manifold import TSNE

# Project the data: this step will take several seconds
tsne = TSNE(n_components=2, random_state=0)
projected_X = tsne.fit_transform(digits.data)

# Compute the clusters
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(projected_X)

# Permute the labels
pred = np.zeros_like(clusters)
for i in range(10):
    mask = (clusters == i)
    pred[mask] = mode(digits.target[mask])[0]
```

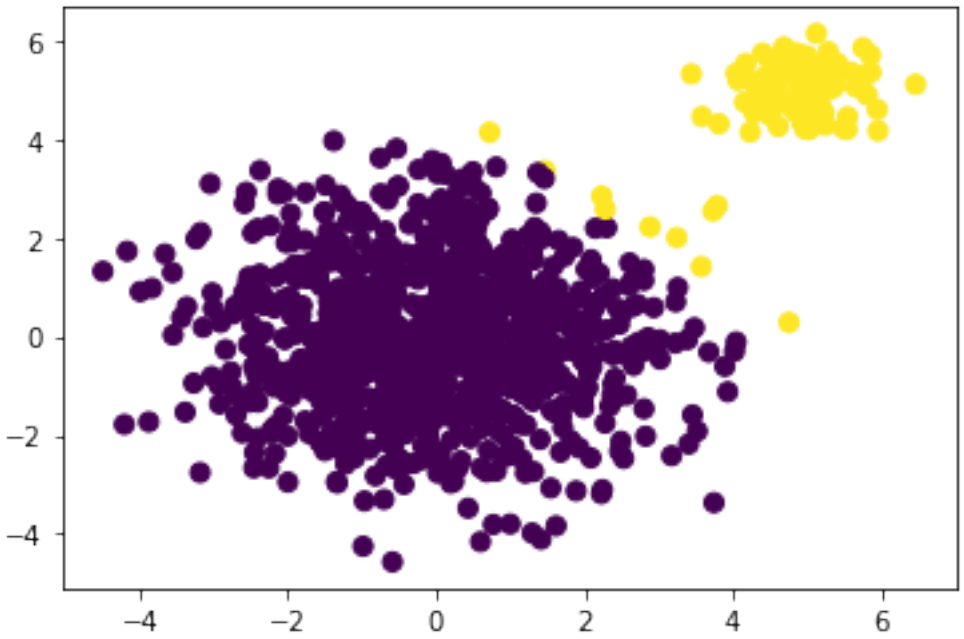
```
# Compute the accuracy
accuracy_score(digits.target, pred)

#yay, after cleaning some noise, we got even higher accuracy!
```

[13]: 0.9371174179187535

K-means does not work well with uneven size clusters

[14]:
```
n_samples_1 = 1000
n_samples_2 = 100
centers = [[0.0, 0.0], [5.0, 5.0]]
clusters_std = [1.5, 0.5]
X, y = make_blobs(n_samples=[n_samples_1, n_samples_2],
                  centers=centers,
                  cluster_std=clusters_std,
                  random_state=0, shuffle=False)

kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')
```

[14]: <matplotlib.collections.PathCollection at 0x126b198d0>



1.2.3 When to use K-means

1. May not guarantee optimal solution. Depends on initialization. Can be fix by running k-means many times with different init random values.
2. Require knowing how many clusters beforehand. Simple way is to use the elbow method which compute within clusters distances. In sklearn, this can be easily computed using kmeans inertia_ variable. For scratch, this is your exercise!
3. Assume spherical distribution. This also means that all k-means assume that clusters have equal number of samples (which may not be true!)
4. Similar to K-nearest neighbors and MDS, k-means can be ridiciously slow for large number of samples. One way to fix this is using the concept of Mini-Batch. It is implemented in sklearn.cluster.MiniBatchMeans. For scratch, this will be another of your exercise!

1.2.4 ===Task===

Your work: Let's modify the above scratch code: - Modify so it print out the total within-cluster variation. Then try to run several k and identify which k is best. - Since k-means can be slow due to its pairwise computations, let's implement a mini-batch k-means in which the cluster is create using only partial subset of samples. - Put everything into a class

1.3 Gaussian Mixture Models

Instead of simply assuming a spherical (circular) shape, we can generalize the Expectation-Maximization algorithm to "weighted sum" of gaussian distribution, we can create a more powerful model - essentially **Gaussian Mixture models**

Under the hood, a Gaussian mixture model is very similar to k-means: it uses an expectation–maximization approach which qualitatively does the following:

1. Choose starting guesses for the location and shape
2. Repeat until converged:
    1. *E-step*: for each point, find weights encoding the probability of membership in each cluster
    2. *M-step*: for each cluster, update its location, normalization, and shape based on *all* data points, making use of the weights

The result of this is that each cluster is associated not with a hard-edged sphere, but with a smooth Gaussian model. Just as in the k-means expectation–maximization approach, this algorithm can sometimes miss the globally optimal solution, and thus in practice multiple random initializations are used.

Mixture models can be used to describe a description $p(x)$ by a convex combination of K simple (base) distributions:

$$p(x) = \sum_{k=1}^{K} \pi_k p_k(x)$$

$$0 \leq \pi_k \leq 1$$

$$\sum_{k=1}^{K} \pi_k = 1$$

where the components p_k are members of a family of basic distributions, e.g., Gaussians, Bernoullis, or Gammas, and the π_k are *mixture weights*. Using these weights allow us to describe datasets with multiple "clusters".

Here we shall focus on **Gaussian** mixture models (GMMs), where the basic distributions are Gaussians. For a given dataset, we aim to maximize the likelihood of the model parammeters to train the GMM.

A *Gaussian mixture model* is a density model where we combine a finite number of K Gaussian distributions

$$\mathcal{N}(x|\mu_k, \Sigma_k)$$

so that

$$p(x|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

$$0 \leq \pi_k \leq 1$$

$$\sum_{k=1}^{K} \pi_k = 1$$

where we define θ as

$$\theta := \{\mu_k, \Sigma_k, \pi_k : k = 1, \cdots, K\}$$

and $\mathcal{N}$ as the multivariate Gaussian distribution, computed using:

$$\mathcal{N}(x|\mu_k, \Sigma_k) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma_k|^{\frac{1}{2}}} \exp(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1}(x - \mu_k))$$

as the collection of all parameters of the model. This convex combination of Gaussian distribution gives us significantly more flexibility for modeling complex densities than a simple Gaussian distribution.

Learning

Assume $X = \{x^{(i)}, \cdots, x^{(m)}\}$ are drawn form an unkown distribution $p(x)$. Our objective is to find a good approximation of this unknown distribution by means of a GMM with K mixture

components. We exploit our i.i.d (independently and identically distributed) assumption, which leads to the log-likelihood as

$$\log p(X|\theta) = \sum_{i=1}^{m} \log p(x^{(i)}|\theta) = \sum_{i=1}^{m} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

Our objective function is to find θ that maximize the log-likehood $\mathcal{L}$

$$\max_{\theta} \sum_{i=1}^{m} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

Our "normal" procedure would be to comute the gradient $\frac{d\mathcal{L}}{d\theta}$ of the log-likelihood with respect to the model parameters θ, set it to 0, and solve for θ, however, if you try this yourself at home, you will find that it is not possible to find the closed form.

One way we can do turns out to be the EM algorithm, where the key idea is to update one model parameter at a time, while keeping the others fixed.

Before we find the partial derivatives, let us introduce a quantity that will play a central role in this algorithm: **responsibilities**.

We define the quantity

$$r_k^{(i)} = \frac{\pi_k \mathcal{N}(x^{(i)} \mid \mu_k, \Sigma_k)}{\Sigma_{j=1}^{K} \pi_j \mathcal{N}(x^{(i)} \mid \mu_j, \Sigma_j)}$$

as the *responsibility* of the kth mixture component for the ith data point.

$r_k^{(i)}$ basically gives us

$$\frac{\text{Probability of } x^{(i)} \text{ belonging to cluster k}}{\text{Probability of } x^{(i)} \text{ over all clusters}}$$

The responsibility $r_k^{(i)}$ of the kth mixture component for data point $x^{(i)}$ is proportional to the likelihood of the mixture component given the data point.

$$p(x^{(i)}|\pi_k, \mu_k, \Sigma_k) = \pi_k \mathcal{N}(x^{(i)}|\mu_k, \Sigma_k)$$

Therefore, mixture components have a high responsibility for a data point when the data point could be a **plausible sample** from that mixture component. Note that

$$r^{(i)} = r_1^{(i)}, r_2^{(i)}, \cdots, r_k^{(i)} \in \mathbb{R}^k$$

is a normalized probability vector, i.e., for each sample i

$$\sum_{j=1}^{k} r_j^{(i)} = 1$$

$$r_j^{(i)} \geq 0$$

Thus this probability vector distributes probability mass among the K mixture components, and we can think of $r^{(ik)}$ as probability that $x^{(i)}$ has been generated by the kth mixture component.

By summing all the total responsibility of the kth mixture component along all samples, we get N_k.

$$N_k = \sum_{i=1}^{m} r_k^{(i)}$$

Note that this value does not necessarily equal to 1.

Updating the mean

The update of the mean parameters $\mu_k, k = 1, \cdots, K$ of the GMM is given by:

$$\mu_k^{new} = \frac{\sum\limits_{i=1}^{m} r_k^{(i)} x^{(i)}}{\sum\limits_{i=1}^{m} r_k^{(i)}}$$

To prove this:

Any local optimum of a function exhibits the property that its gradient with respect to the parameters must vanish, i.e., setting its partial derivative to zero.

We take a partial derivative of our objective function with respect to the mean parameters $\mu_k, k = 1, \cdots, K$. To simplify things, let's perform partial derivative without the log first and only consider one sample.

$$\frac{\partial p(x^{(i)}|\theta)}{\partial \mu_k} = \sum_{j=1}^{K} \pi_j \frac{\partial \mathcal{N}(x^{(i)}|\mu_j, \Sigma_j)}{\partial \mu_k} = \pi_k \frac{\partial \mathcal{N}(x^{(i)}|\mu_k, \Sigma_k)}{\partial \mu_k} = \pi_k (x^{(i)} - \mu_k)^T \Sigma_k^{-1} \mathcal{N}(x^{(i)}|\mu_k, \Sigma_k)$$

Now, taking all samples and log, since we know the partial derivative of log something is $\frac{1}{x}$, thus

$$\frac{\partial \mathcal{L}}{\partial \mu_k} = \sum_{i=1}^{m} \frac{\partial \log p(x^{(i)}|\theta)}{\partial \mu_k} = \sum_{i=1}^{m} \frac{1}{p(x^{(i)}|\theta)} \frac{\partial p(x^{(i)}|\theta)}{\partial \mu_k} = \sum_{i=1}^{m} (x^{(i)} - \mu_k)^T \Sigma_k^{-1} \frac{\pi_k \mathcal{N}(x^{(i)} \mid \mu_k, \Sigma_k)}{\Sigma_{j=1}^{K} \pi_j \mathcal{N}(x^{(i)} \mid \mu_j, \Sigma_j)}$$

To simplify, we can substitute $r_k^{(i)}$ into the equation, thus

$$= \sum_{i=1}^{m} r_k^{(i)} (x^{(i)} - \mu_k)^T \Sigma_k^{-1}$$

We can now solve for μ_k so that $\frac{\partial \mathcal{L}}{\partial \mu_k} = 0$ and obtain

$$\sum_{i=1}^{m} r_k^{(i)} (x^{(i)} - \mu_k)^T \Sigma_k^{-1} = 0$$

Multiply both sides by Σ will cancel out the inverse Σ, and move μ_k to another side

$$\sum_{i=1}^{m} r_k^{(i)} x^{(i)} = \sum_{i=1}^{m} r_k^{(i)} \mu_k$$

$$\frac{\sum_{i=1}^{m} r_k^{(i)} x^{(i)}}{\sum_{i=1}^{m} r_k^{(i)}} = \mu_k$$

We can further substitute N_k so that

$$\frac{1}{N_k} \sum_{i=1}^{m} r_k^{(i)} x^{(i)} = \mu_k$$

Here we can interpret that μ_k is pulled toward a data point $x^{(i)}$ with strength given by $r_k^{(i)}$. The means are pulled stronger toward data points for which the corresponding mixture component has a high responsibility, i.e., a high likelihood.

Updating the covariances

The update of the covariance parameters $\Sigma_k, k = 1, \cdots, K$ of the GMM is given by:

$$\Sigma_k^{new} = \frac{1}{N_k} \sum_{i=1}^{m} r_k^{(i)} (x^{(i)} - \mu_k)(x^{(i)} - \mu_k)^T$$

To prove this:

We take a partial derivative of our objective function with respect to the Sigma parameters $\Sigma_k, k = 1, \cdots, K$. Similarly, to simplify things, let's perform partial derivative without the log first and only consider one sample.

$$\frac{\partial p(x^{(i)}|\theta)}{\partial \Sigma_k} = \frac{\partial}{\partial \Sigma_k} \big(\pi_k (2\pi)^{-\frac{D}{2}} \det(\Sigma_k)^{\frac{1}{2}} exp\big(-\frac{1}{2} (x^{(i)} - \mu_k)^T \Sigma_k^{-1} (x^{(i)} - \mu_k) \big) \big)$$

Using derivative multiplication rule, we got

$$= \pi_k (2\pi)^{-\frac{D}{2}} \big[\frac{\partial}{\partial \Sigma_k} \det(\Sigma_k)^{-\frac{1}{2}} exp\big(-\frac{1}{2} (x^{(i)} - \mu_k)^T \Sigma_k^{-1} (x^{(i)} - \mu_k) \big) + \det(\Sigma_k)^{-\frac{1}{2}} \frac{\partial}{\partial \Sigma_k} exp\big(-\frac{1}{2} (x^{(i)} - \mu_k)^T \Sigma_k^{-1} (x^{(i)} - \mu_k) \big]$$

Using this following rule

$$\frac{\partial}{\partial X}\det(f(x)) = \det(f(x))tr\big(f(x)^{-1}\frac{\partial f(x)}{\partial x}\big)$$

We get that

$$\frac{\partial}{\partial \Sigma_k}\det(\Sigma_k)^{-\frac{1}{2}} = -\frac{1}{2}\det(\Sigma_k)^{-\frac{1}{2}}\Sigma_k^{-1}$$

Using this following rule

$$\frac{\partial a^T X b}{\partial X} = ab^T$$

We get that

$$\frac{\partial}{\partial \Sigma_k}(x^{(i)}-\mu_k)^T\Sigma_k^{-1}(x^{(i)}-\mu_k) = -\Sigma_k^{-1}(x^{(i)}-\mu_k)(x^{(i)}-\mu_k)^T\Sigma_k^{-1}$$

Putting them together, we got:

$$\frac{\partial p(x^{(i)}|\theta)}{\partial \Sigma_k} = \pi_k\mathcal{N}(x^{(i)}|\mu_k,\Sigma_k) * \big[-\frac{1}{2}(\Sigma_k^{-1} - \Sigma_k^{-1}(x^{(i)}-\mu_k)(x^{(i)}-\mu_k)^T\Sigma_k^{-1}\big]$$

Now consider all samples and log as well, the partial derivative of the log-likelihood with respect to Σ_k is given by

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \Sigma_k} &= \sum_{i=1}^{m}\frac{\partial \log p(x^{(i)}|\theta)}{\partial \Sigma_k} \\
&= \sum_{i=1}^{m}\frac{1}{(p(x^{(i)}|\theta)}\frac{\partial p(x^{(i)}|\theta)}{\partial \Sigma_k} \\
&= \sum_{i=1}^{m}\frac{\pi_k\mathcal{N}(x^{(i)} \mid \mu_k,\Sigma_k)}{\Sigma_{j=1}^{K}\pi_j\mathcal{N}(x^{(i)} \mid \mu_j,\Sigma_j)} * \big[-\frac{1}{2}(\Sigma_k^{-1} - \Sigma_k^{-1}(x^{(i)}-\mu_k)(x^{(i)}-\mu_k)^T\Sigma_k^{-1})\big]
\end{aligned}$$

Substituting $r_k^{(i)}$, we got

$$= -\frac{1}{2}\sum_{i=1}^{m} r_k^{(i)}(\Sigma_k^{-1} - \Sigma_k^{-1}(x^{(i)}-\mu_k)(x^{(i)}-\mu_k)^T\Sigma_k^{-1}) = -\frac{1}{2}\Sigma_k^{-1}\sum_{i=1}^{m} r_k^{(i)} + \frac{1}{2}\Sigma_k^{-1}\big(\sum_{i=1}^{m} r_k^{(i)}(x^{(i)}-\mu_k)(x^{(i)}-\mu_k)^T\big)\Sigma_k^{-1}$$

Setting this to zero, we obtain:

$$N_k\Sigma_k^{-1} = \Sigma_k^{-1}\big(\sum_{i=1}^{m} r_k^{(i)}(x^{(i)}-\mu_k)(x^{(i)}-\mu_k)^T\big)\Sigma_k^{-1}$$

By solving for Σ_k we got

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{m} r_k^{(i)} (x^{(i)} - \mu_k)(x^{(i)} - \mu_k)^T$$

Updating the pi - weight of mixture components

The update of the mixture weights $\pi_k, k = 1, \cdots, K$ of the GMM is given by:

$$\pi_k^{new} = \frac{N_k}{m}$$

To prove this:

To find the partial derivative, we account for the equality constraint

$$\sum_{k=1}^{K} \pi_k = 1$$

The Lagrangian $\mathscr{L}$ is

$$\begin{aligned} \mathscr{L} &= \mathcal{L} + \beta\big(\sum_{k=1}^{K} \pi_k - 1\big) \\ &= \sum_{i=1}^{m} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) + \beta\big(\sum_{k=1}^{K} \pi_k - 1\big) \end{aligned}$$

Taking the partial derivative with respect to π_k as

$$\begin{aligned} \frac{\partial \mathscr{L}}{\partial \pi_k} &= \sum_{i=1}^{m} \frac{\mathcal{N}(x^{(i)} \mid \mu_k, \Sigma_k)}{\Sigma_{j=1}^{K} \pi_j \mathcal{N}(x^{(i)} \mid \mu_j, \Sigma_j)} + \beta \\ &= \frac{1}{\pi_k} \sum_{i=1}^{m} \frac{\pi_k \mathcal{N}(x^{(i)} \mid \mu_k, \Sigma_k)}{\Sigma_{j=1}^{K} \pi_j \mathcal{N}(x^{(i)} \mid \mu_j, \Sigma_j)} + \beta \\ &= \frac{N_k}{\pi_k} + \beta \end{aligned}$$

Taking the partial derivative with respect to β is

$$\frac{\partial \mathscr{L}}{\partial \beta} = \sum_{i=1}^{K} \pi_k - 1$$

Setting both partial derivatives to zero yield

$$\pi_k = -\frac{N_k}{\beta}$$

$$1 = \sum_{i=1}^{K} \pi_k$$

Using the top formula to solve for the bottom formula:

$$-\sum_{i=1}^{m} \frac{N_k}{\beta} = 1 = -\frac{m}{\beta} = 1 = \beta = -m$$

Substitute $-m$ for β yield

$$\pi_k = \frac{N_k}{m}$$

Algorithm

Thus, we can summarize the whole algorithm into the following steps:

1. Define k number of clusters c
2. For each cluster k, randomly initialize parameters mean μ_k, covariance Σ_k, and fraction per class π_k
3. *E-step:* Evaluate responsibilities $r_k^{(i)}$ for every data point $x^{(i)}$ using

$$r_k^{(i)} = \frac{\pi_k \mathcal{N}(x^{(i)} \mid \mu_k, \Sigma_k)}{\Sigma_{j=1}^{K} \pi_j \mathcal{N}(x^{(i)} \mid \mu_j, \Sigma_j)}$$

4. *M-step:* Restimate parameters π_k, μ_k, Σ_k using the current responsibilites $r_k^{(i)}$ from the E step.

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{m} r_k^{(i)} x^{(i)}$$

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{m} r_k^{(i)} (x^{(i)} - \mu_k)(x^{(i)} - \mu_k)^T$$

$$\pi_k = \frac{N_k}{m}$$

Coding considerations

1. To ease our programming efforts, we can use scipy.stats.multivariate_normal for generating gaussian distribution, and using its .pdf() function to compute the values we want of $N(x_i \mid \mu_k, \Sigma_k)$
2. Shape of r which keeps probability of $x^{(i)}$ belonging to kth cluster is $(m * k)$, where m is X.shape[0] and k is number of clusters we want.

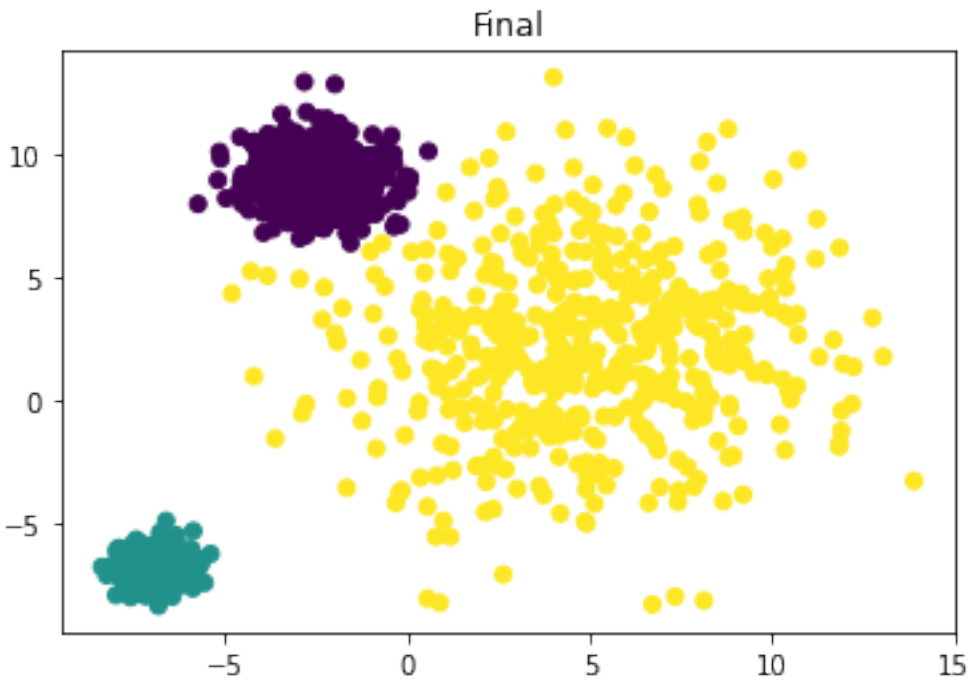
3. Shape of Σ or covariance of each cluster is simply $(n * n)$ where n is number of features or X.shape[1]. If we define 3 clusters, then we will have $[\Sigma_1, \Sigma_2, \Sigma_3]$, each with shape $(n * n)$, thus whole thing is shape $ (k * n * n) $
4. Shape of π is simply $(k,)$
5. Shape of μ is (n, k), defining mean for each feature for kth cluster
6. What to initialize can be tricky. For r and π, you can fill with $1/k$. As for μ, it is easiest to simply pick random points from the samples as initial means. Last, for covariance (Σ), we can simply use the covariance of the X as initialization (i.e., np.cov(X.T))

1.3.1 Scratch

[15]:
```
from sklearn.cluster import KMeans
from scipy.stats import multivariate_normal
import math

X, y = make_blobs(n_samples=1500, cluster_std=[1.0, 3.5, 0.5], random_state=42)

#define basic params
m, n = X.shape
K = 3
max_iter = 20

#==initialization==

#responsibliity
r = np.full(shape=(m, K), fill_value=1/K)

#pi
pi = np.full((K, ), fill_value=1/K) #simply use 1/k for pi

#mean
random_row = np.random.randint(low=0, high=m, size=K)
mean = np.array([X[idx,:] for idx in random_row ]).T #.T to make to shape (M, K)

#covariance
cov = np.array([np.cov(X.T) for _ in range (K)])

for iteration in range(max_iter):

    #===E-Step=====
    #Update r_ik of each sample
    for i in range(m):
        for k in range(K):
            xi_pdf = multivariate_normal.pdf(X[i], mean=mean[:, k], cov=cov[k])
            r[i, k] = pi[k] * xi_pdf
        r[i] /= np.sum(r[i])
```

```
    #===M-Step====
    # Find NK first for latter use
    NK = np.sum(r, axis=0)
    assert NK.shape == (K, )

    #PI
    pi = NK / m
    assert pi.shape == (K, )

    #mean
    mean =  ( X.T @ r ) / NK
    assert mean.shape == (n, K)

    #covariance (also called Sigma)
    cov = np.zeros((K, n, n))
    for k in range(K):
        for i in range(m):
            X_mean = (X[i]-mean[:, k]).reshape(-1, 1)
            cov[k] += r[i, k] * (X_mean @ X_mean.T)
        cov[k] /= NK[k]
    assert cov.shape == (K, n, n)


#get preds
yhat = np.argmax(r, axis=1)

#plot
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=yhat)
plt.title("Final")
```

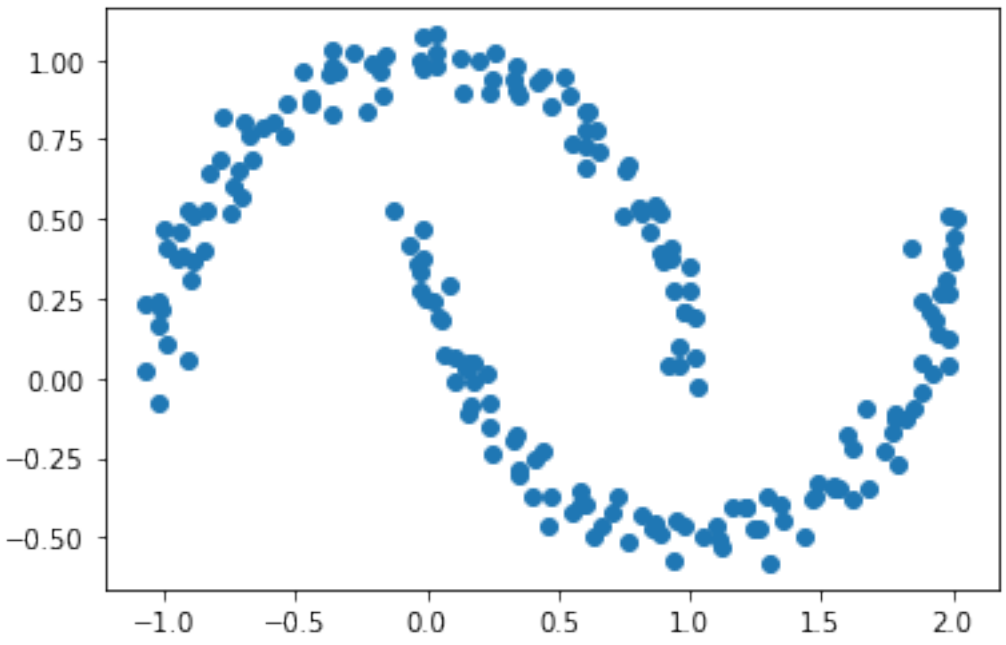
[15]: Text(0.5, 1.0, 'Final')



1.3.2 Sklearn

Though GMM is often categorized as a clustering algorithm, fundamentally it is an algorithm for *density estimation*. That is to say, the result of a GMM fit to some data is technically not a clustering model, but a generative probabilistic model describing the distribution of the data.

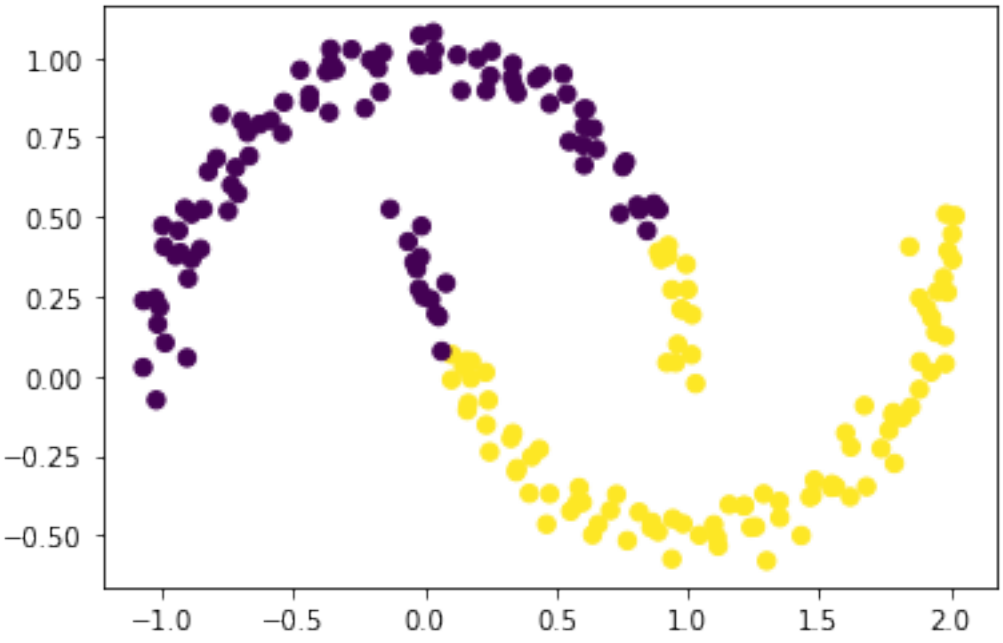
[16]:
```
from sklearn.datasets import make_moons
Xmoon, ymoon = make_moons(200, noise=.05, random_state=0)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```



If we try to fit this with a two-component GMM viewed as a clustering model, the results are not particularly useful:

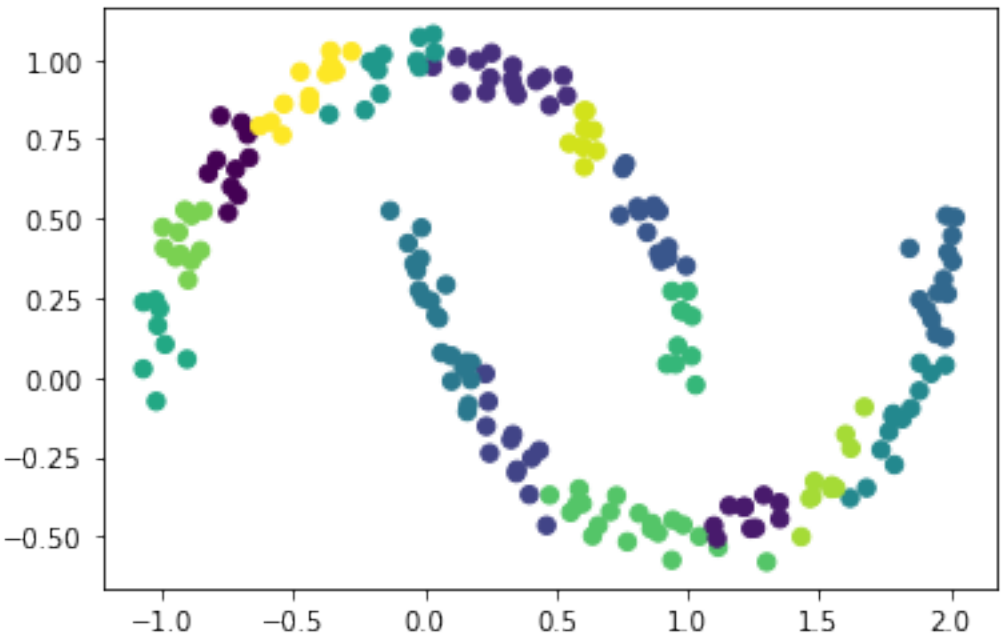
[17]:
```
from sklearn.mixture import GaussianMixture as GMM
gmm2 = GMM(n_components=2, covariance_type='full', random_state=0)
pred = gmm2.fit(Xmoon).predict(Xmoon)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1], c=pred, s=40, cmap='viridis')
```

[17]:
```
<matplotlib.collections.PathCollection at 0x126b01110>
```



[18]:
```
gmm16 = GMM(n_components=16, covariance_type='full', random_state=0)
pred = gmm16.fit(Xmoon).predict(Xmoon)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1], c=pred, s=40, cmap='viridis')
```

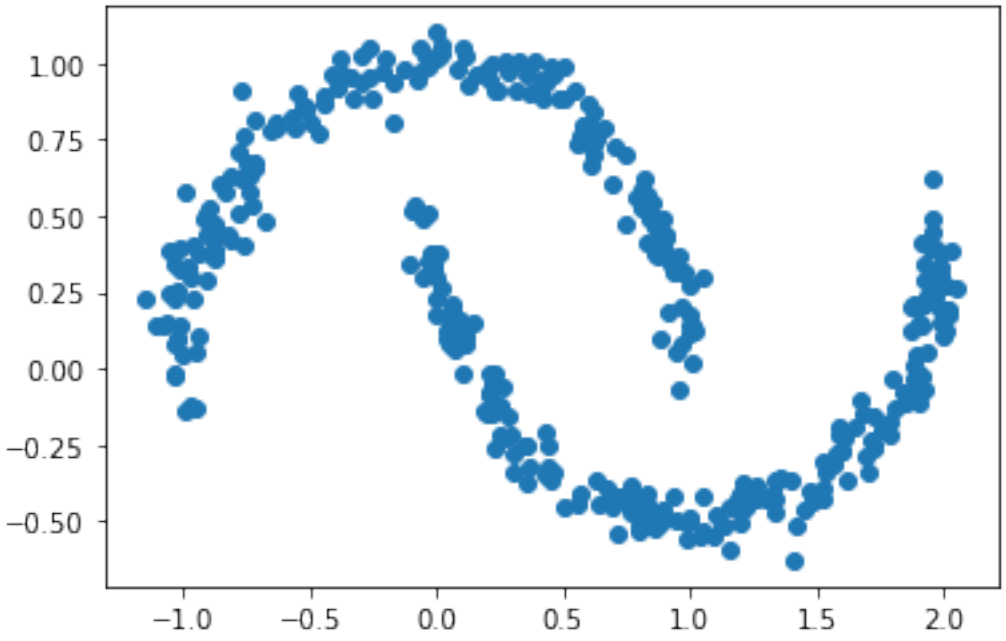
[18]: <matplotlib.collections.PathCollection at 0x126bd7ed0>



Here the mixture of 16 Gaussians serves not to find separated clusters of data, but rather to model the overall *distribution* of the input data. This is a generative model of the distribution, meaning that the GMM gives us the recipe to generate new random data distributed similarly to our input. For example, here are 400 new points drawn from this 16-component GMM fit to our original data:

[19]:
```
Xnew, _ = gmm16.sample(400)
plt.scatter(Xnew[:, 0], Xnew[:, 1])
```

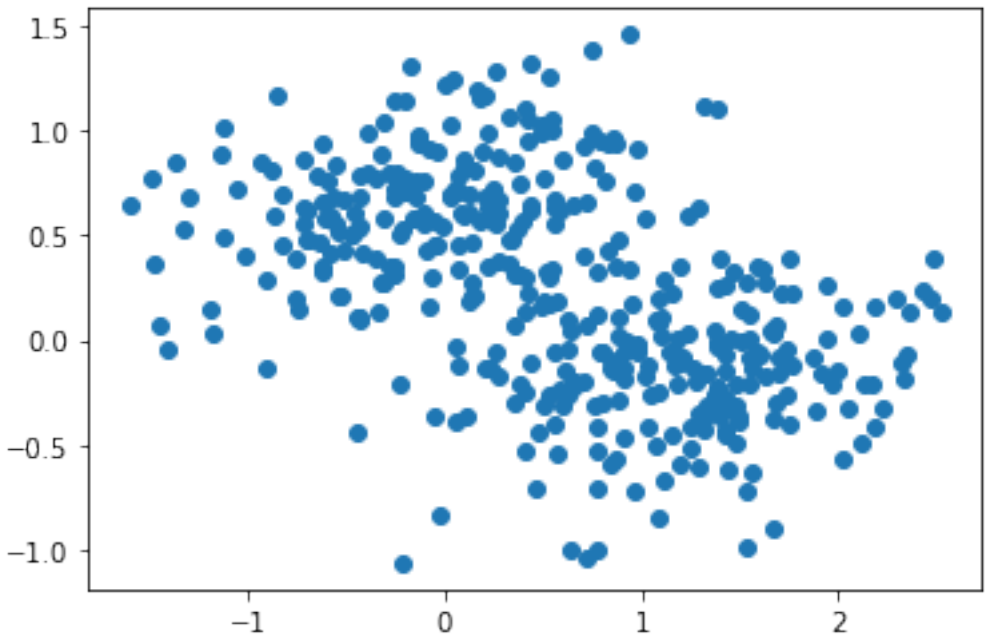
[19]:
```
<matplotlib.collections.PathCollection at 0x12ac77250>
```



The above data is a "generated" data - not real data. So by understanding the distribution, it is useful to generate the data

[20]:
```
#Let's also take a look on gmm2.sample
Xnew, _ = gmm2.sample(400)
plt.scatter(Xnew[:, 0], Xnew[:, 1])
```

[20]:
```
<matplotlib.collections.PathCollection at 0x12ae31c10>
```



How many components for understanding distribution? The fact that GMM is a generative model gives us a natural means of determining the optimal number of components for a given dataset. A generative model is inherently a probability distribution for the dataset, and so we can simply evaluate the *likelihood* of the data under the model, using cross-validation to avoid over-fitting.

Another means of correcting for over-fitting is to adjust the model likelihoods using some analytic criterion such as the Akaike information criterion (AIC) or the Bayesian information criterion (BIC)

$$AIC = 2k - 2ln(L)$$

$$BIC = kln(n) - 2ln(L)$$

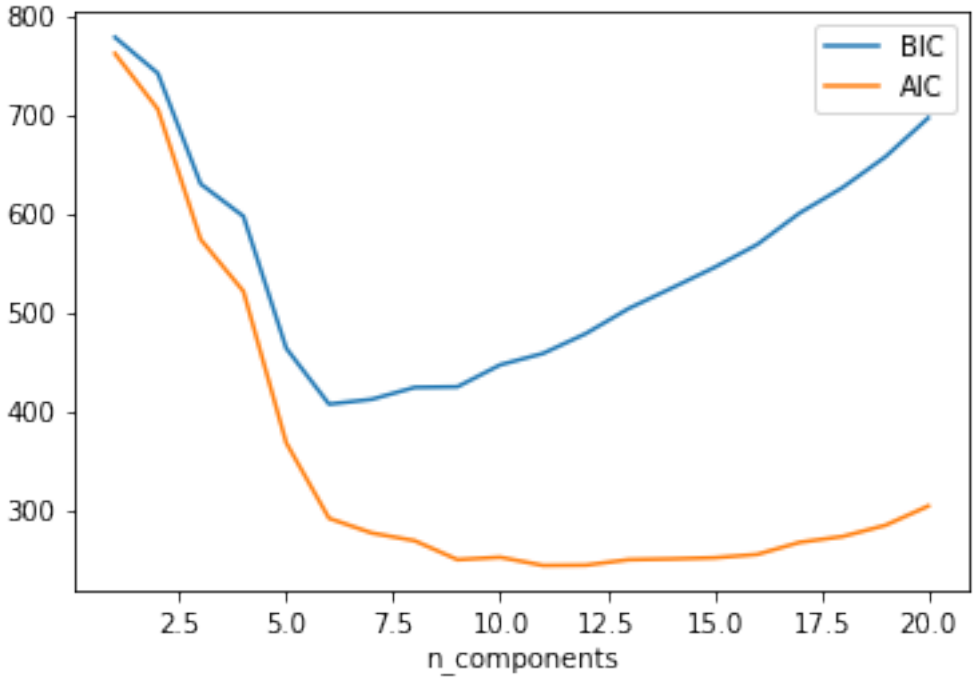
where k is the number of features and L is maximum likelihood. Basically, number of features will increase AIC, while L will decrease AIC. We want to good balance between model complexity and goodness of fit, thus lower the AIC, the better.

Scikit-Learn's `GMM` estimator actually includes built-in methods that compute AIc, and so it is very easy to operate on this approach.

Let's look at the AIC as a function as the number of GMM components for our moon dataset:

[21]:
```
n_components = np.arange(1, 21)
models = [GMM(n, random_state=0).fit(Xmoon)
          for n in n_components]
```

```
plt.plot(n_components, [m.bic(Xmoon) for m in models], label='BIC')
plt.plot(n_components, [m.aic(Xmoon) for m in models], label='AIC')
plt.legend(loc='best')
plt.xlabel('n_components');
```



The optimal number of clusters is the value that minimizes the AIC or BIC, depending on which approximation we wish to use. The AIC tells us that our choice of 16 components above was probably too many: around 8-12 components would have been a better choice. As is typical with this sort of problem, the BIC recommends a simpler model.

Notice the important point: this choice of number of components measures how well GMM works *as a density estimator*, not how well it works *as a clustering algorithm*. I'd encourage you to think of GMM primarily as a density estimator, and use it for clustering only when warranted within simple datasets.

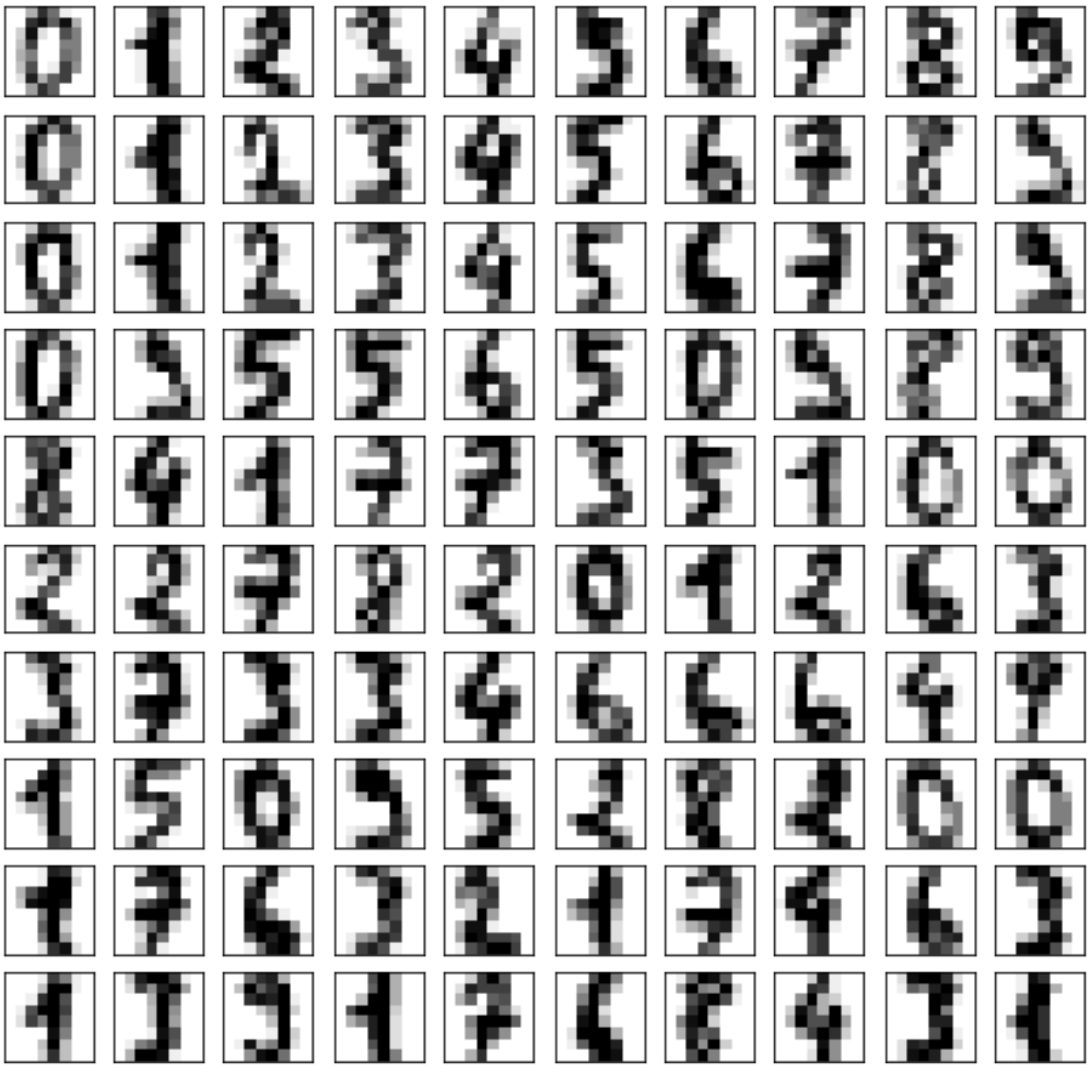
Example: GMM for Generating New Data Here, since GMM is kind of model that tries to understand the data probability distribution. GMM can be used to generate *new handwritten digits* from the standard digits corpus that we have used before.

To start with, let's load the digits data using Scikit-Learn's data tools:

[23]:
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

[23]: (1797, 64)

[24]:
```
def plot_digits(data):
    fig, ax = plt.subplots(10, 10, figsize=(8, 8),
                           subplot_kw=dict(xticks=[], yticks=[]))
    for i, axi in enumerate(ax.flat):
        im = axi.imshow(data[i].reshape(8, 8), cmap='binary')
        im.set_clim(0, 16)
plot_digits(digits.data)
```



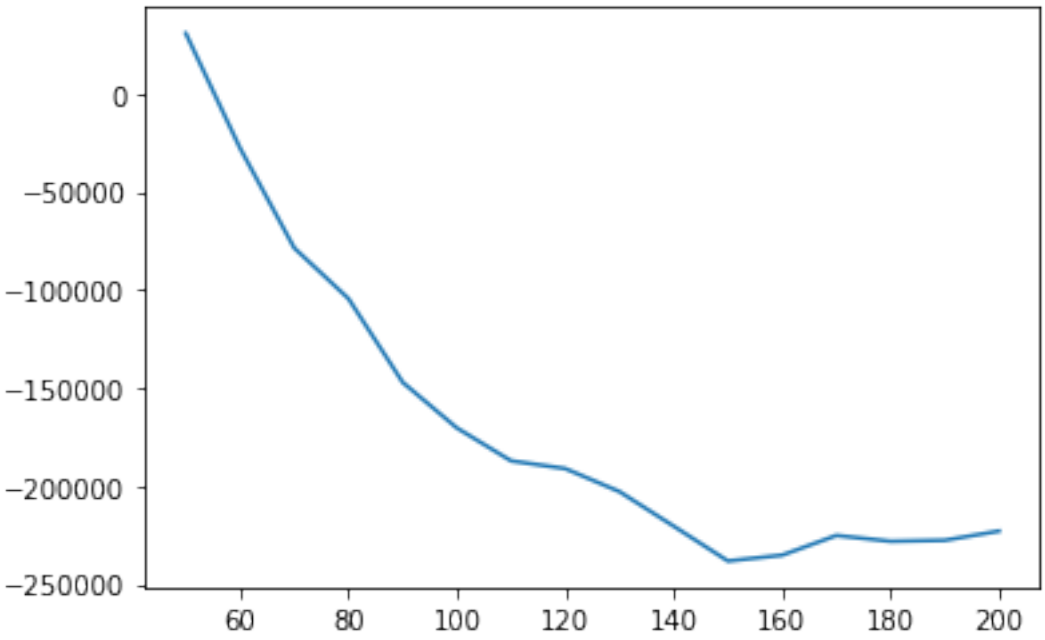
We have nearly 1,800 digits in 64 dimensions, and we can build a GMM on top of these to generate more. GMMs can have difficulty converging in such a high dimensional space, so we will start with an invertible dimensionality reduction algorithm on the data. Here we will use a straightforward PCA, asking it to preserve 99% of the variance in the projected data:

[25]:
```
from sklearn.decomposition import PCA
pca = PCA(0.99, whiten=True)
data = pca.fit_transform(digits.data)
data.shape  #41 dimensions
```

[25]:
```
(1797, 41)
```

The result is 41 dimensions, a reduction of nearly 1/3 with almost no information loss. Given this projected data, let's use the AIC to get a gauge for the number of GMM components we should use:

[26]:
```
n_components = np.arange(50, 210, 10)
models = [GMM(n, random_state=0)
          for n in n_components]
aics = [model.fit(data).aic(data) for model in models]
plt.plot(n_components, aics);
```



It appears that around 110 components minimizes the AIC; we will use this model. Let's quickly fit this to the data and confirm that it has converged:

[27]:
```
gmm = GMM(110, covariance_type='full', random_state=0)
gmm.fit(data)
print(gmm.converged_)
```

```
True
```

Now we can draw samples of 100 new points within this 41-dimensional projected space, using the

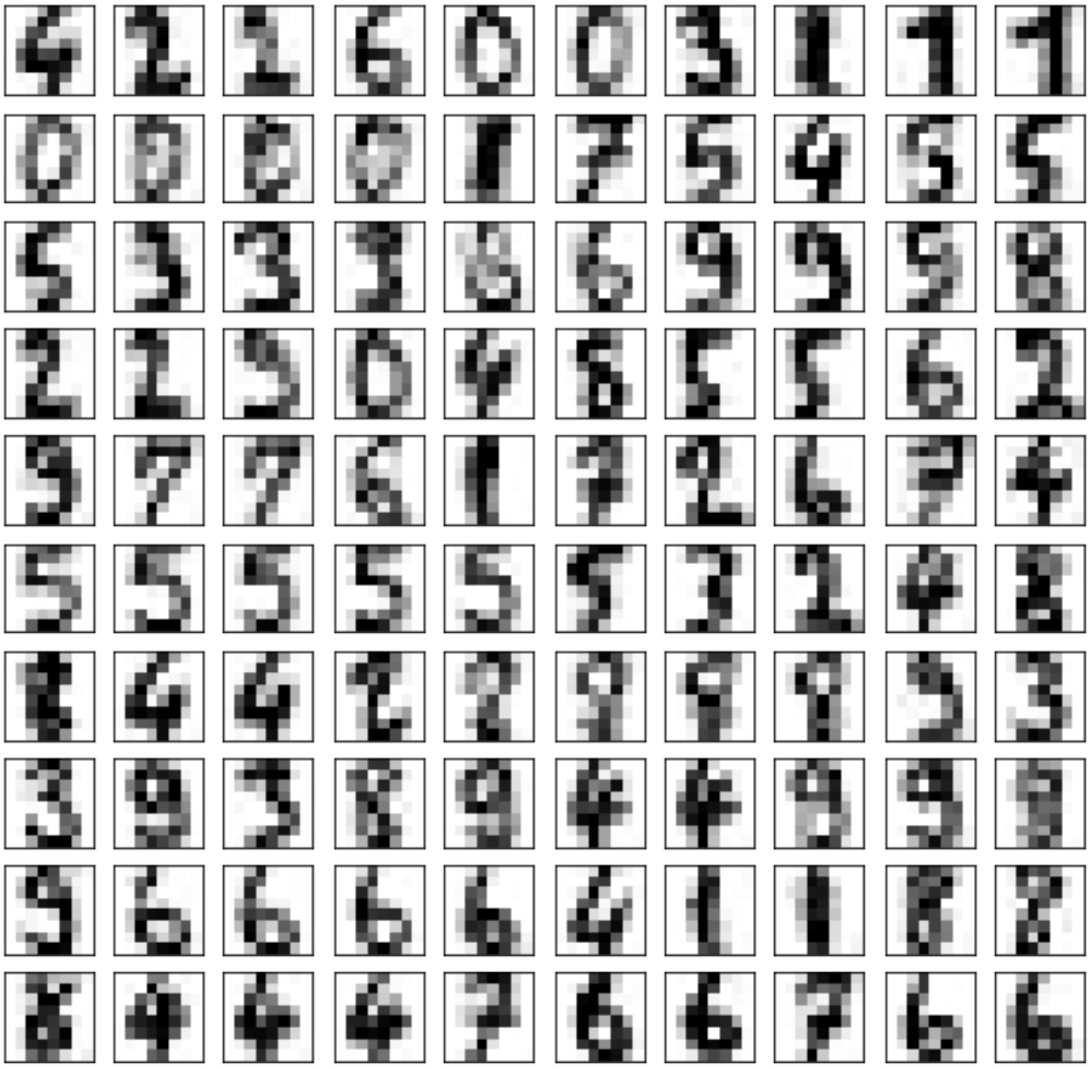
GMM as a generative model:

[28]:
```
X, y = gmm.sample(100)
X.shape
```

[28]:
```
(100, 41)
```

Finally, we can use the inverse transform of the PCA object to construct the new digits:

[29]:
```
digits_new = pca.inverse_transform(X)
plot_digits(digits_new)
```



Consider what we've done here: given a sampling of handwritten digits, we have modeled the distribution of that data in such a way that we can generate brand new samples of digits from the data: these are "handwritten digits" which do not individually appear in the original dataset, but

rather capture the general features of the input data as modeled by the mixture model. Such a generative model of digits is perhaps the basic idea behind **"Generative Adversarial Network"**

1.3.3 ===Task===

Your work: Let's modify the above scratch code: - Modify so it performs early stopping when the log likelihood does not improve anymore. - Perform plotting every 5 iterations on the resulting clusters.

[ ]: