# K-Means

In [ ]:
```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

In [2]:
```
%run kmeans
```

The k-means algorithm searches for a pre-determined number of clusters within an unlabeled multidimensional dataset.

Algorithm:

The way kmeans algorithm works is as follows:

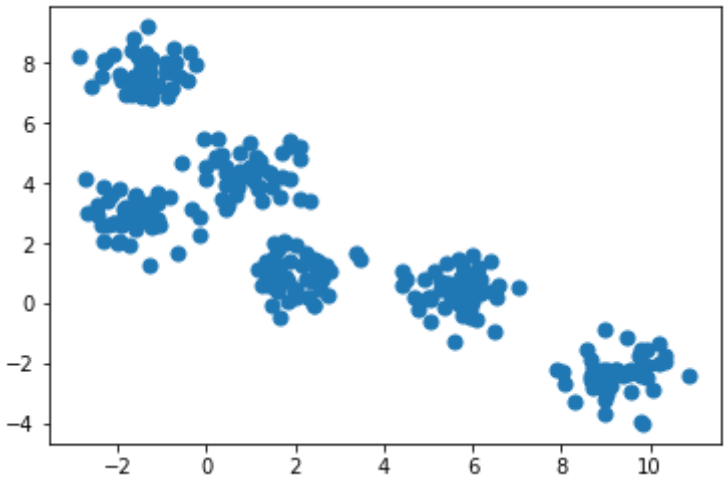
1.> Specify number of clusters K.
2.> Initialize centroids by randomly selecting K data points for the centroids without replacement.
3.> Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing.
   a.>Compute the sum of the squared distance between data points and all centroids.
   b.>Assign each data point to the closest cluster (centroid).
   c.>Recompute the centroids for the clusters by taking the average of the all data points that belong to each cluster.

The approach kmeans follows to solve the problem is called **Expectation-Maximization**. The E-step is assigning the data points to the closest cluster. The M-step is computing the centroid of each cluster

## Kmeans demonstration

First, let's generate a two-dimensional dataset containing six distinct blobs

In [12]:
```
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=300, centers=6,
                       cluster_std=0.60, random_state=0)
plt.scatter(X[:, 0], X[:, 1], s=50);
```

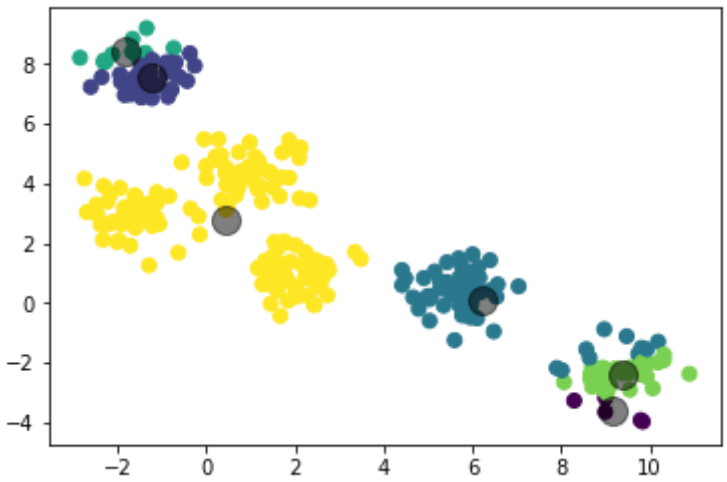


In [13]:
```
k=6
centroids_X, clusters_X = kmeans(X, k=k, tolerance=.01)
```

In [14]:
```
y_kmeans=np.zeros(300)
for i in range(k):
        y_kmeans[clusters_X[i]]=i
```

In [15]:
```
plt.scatter(X[:, 0], X[:, 1] ,c=y_kmeans, s=50, cmap='viridis')
plt.scatter(centroids_X[:, 0], centroids_X[:, 1], c='black', s=200,
            alpha=0.5)
```

Out[15]:
```
<matplotlib.collections.PathCollection at 0x124be0e80>
```



One disadvantage of the K-means algorithm is that it is sensitive to the initialization of the centroids.
So, if a centroid is initialized to be a “far-off” point, it might just end up with no points associated

with it and at the same time more than one clusters might end up linked with a single centroid. Similarly, more than one centroids might be initialized into the same cluster resulting in poor clustering.

To overcome the above-mentioned drawback we use K-means++. This algorithm ensures a smarter initialization of the centroids and improves the quality of the clustering. Apart from initialization, the rest of the algorithm is the same as the standard K-means algorithm.

## kmeans++

Initialization algorithm:

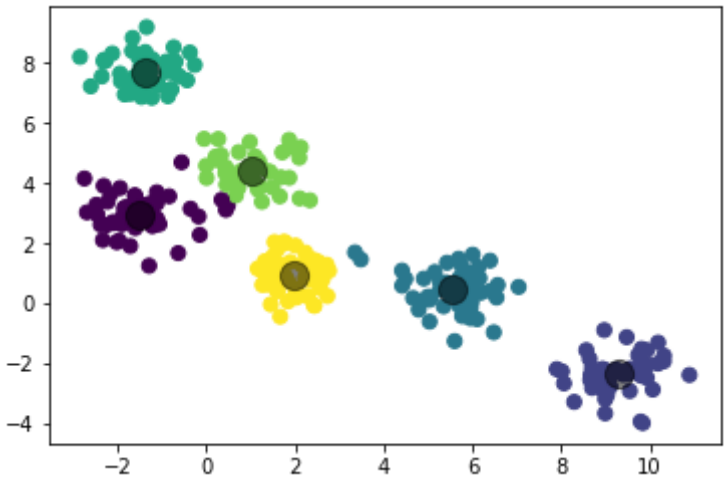
1.> Randomly select the first centroid from the data points.
2.> For each data point compute its distance from the nearest, previously choosen centroid.
3.>Select the next centroid from the data points such that the probability of choosing a point as centroid is directly proportional to its distance from the nearest, previously chosen centroid. (i.e. the point having maximum distance from the nearest centroid is most likely to be selected next as a centroid)
4.>Repeat steps 2 and 3 untill k centroids have been sampled

In [16]:
```
k=6
centroids_X, clusters_X = kmeans(X, k=k, centroids='kmeans++',
                                 tolerance=.01)
```

In [17]:
```
y_kmeans=np.zeros(300)
for i in range(k):
        y_kmeans[clusters_X[i]]=i
```

In [18]:
```
plt.scatter(X[:, 0], X[:, 1] ,c=y_kmeans, s=50, cmap='viridis')
plt.scatter(centroids_X[:, 0], centroids_X[:, 1], c='black', s=200,
            alpha=0.5)
```

Out[18]:
```
<matplotlib.collections.PathCollection at 0x124ce35c0>
```




Note: Although the initialization in K-means++ is computationally more expensive than the standard

K-means algorithm, the run-time for convergence to optimum is drastically reduced for K-means++. This is because the centroids that are initially chosen are likely to lie in different clusters already.

# Spectral clustering

**Drawback on k-means clustering:**

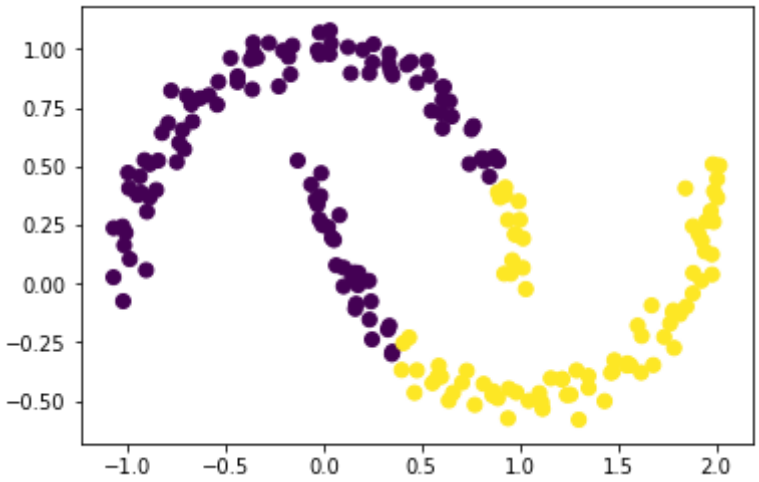
Kmeans algorithm is good in capturing structure of the data if clusters have a spherical-like shape. It always try to construct a nice spherical shape around the centroid. That means, the minute the clusters have a complicated geometric shapes, kmeans does a poor job in clustering the data.

In [41]:
```
from sklearn.datasets import make_moons
X, y = make_moons(200, noise=.05, random_state=0)
```

In [42]:
```
k=2
centroids, clusters = kmeans(X, k=k, centroids='kmeans++',
                             tolerance=.01)
```

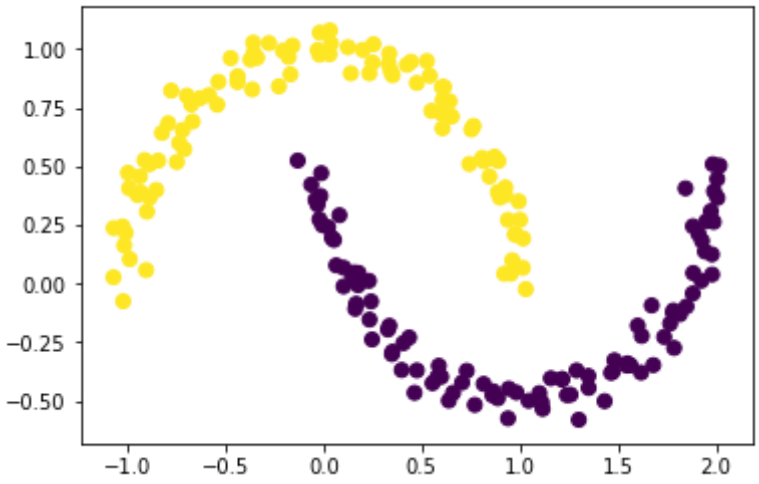
In [44]:
```
y_kmeans=np.zeros(200)
for i in range(k):
        y_kmeans[clusters[i]]=i
```

In [45]:
```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis');
```




**Spectral Clustering work very well in such scenarios**

In [61]:
```
from sklearn.cluster import SpectralClustering
model = SpectralClustering(n_clusters=2, affinity='nearest_neighbors',
                           assign_labels='kmeans')
labels = model.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```




# Applications of k-means clustering

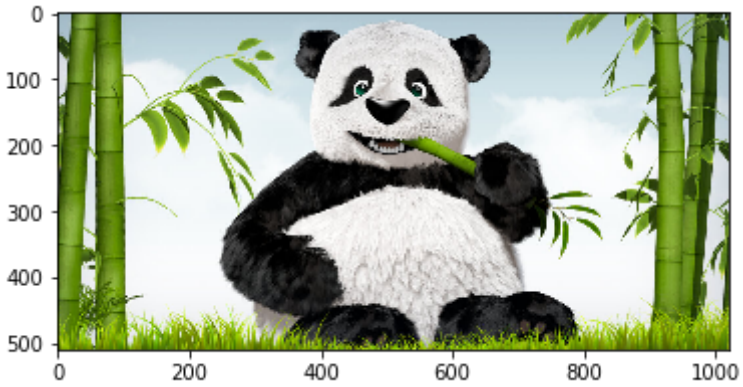
## Compressing a coloured image

In this part, we'll implement kmeans to compress an image.

For each pixel location we would have 3 8-bit integers that specify the red, green, and blue intensity values. Our goal is to reduce the number of colors to 32 and represent (compress) the photo using those 32 colors only. To pick which colors to use, we'll use kmeans algorithm on the image and treat every pixel as a data point.

In [71]:
```
img = mpimg.imread('website.jpg')
```

In [73]:
```
imgplot = plt.imshow(img)
```




In [64]:
```
image_reshaped_2D = np.reshape(img,
                               (img.shape[0] * img.shape[1], img.shape[2]))
```

In [7]:
```
k1=32
centroids, clusters = kmeans(image_reshaped_2D, k=k1,
                             centroids='kmeans++', tolerance=.01)
centroids = centroids.astype(np.uint8)
```

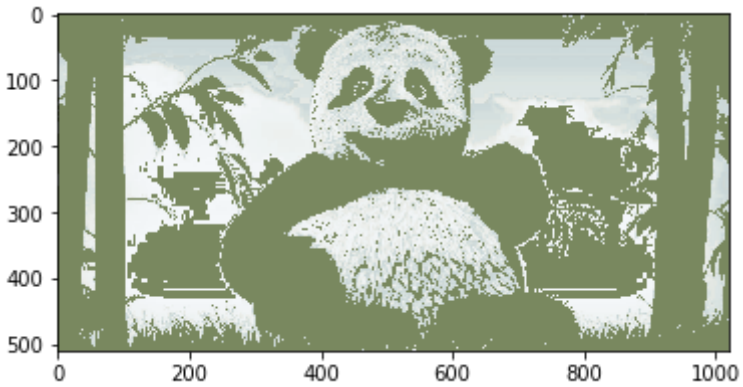
In [8]:
```
def reassign_colors(X, centroids, clusters, k):
    m, n = X.shape
    ret_X=np.zeros((m, n))
    for i in range(k):
        ret_X[clusters[i]]=centroids[i]
    return ret_X
```

In [9]:
```
recovered_img = reassign_colors(image_reshaped_2D,
                                centroids, clusters, k1)
```

In [10]:
```
recovered_img_3D = np.reshape(recovered_img,
                              (img.shape[0], img.shape[1], img.shape[2]))
```

In [11]:
```
plt.imshow((recovered_img_3D).astype(np.uint8))
```

Out[11]:
```
<matplotlib.image.AxesImage at 0x1263fa438>
```




## k-means on digits

Here we will attempt to use k-means to try to identify similar digits without using the original label information

We will start by loading the digits and then finding the KMeans clusters. 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:

In [175]:
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

Out[175]:
```
(1797, 64)
```

In [176]:
```
k=10
centroids, clusters = kmeans(digits.data, k=k,
                             centroids='kmeans++', tolerance=.01)
```

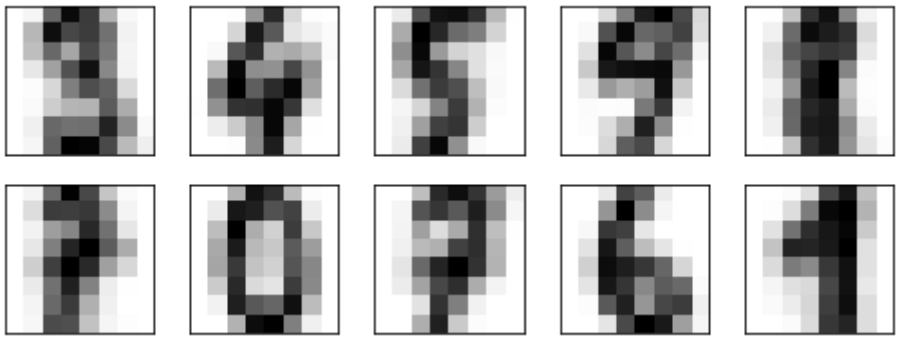
In [177]:
```
centroids.shape
```

Out[177]:
```
(10, 64)
```

In [178]:
```
y_kmeans=np.zeros(1797)
for i in range(k):
    y_kmeans[clusters[i]]=i
```

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:

In [179]:
```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = centroids.reshape(10, 8, 8)
for axi, center in zip(ax.flat, centers):
    axi.set(xticks=[], yticks=[])
    axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```



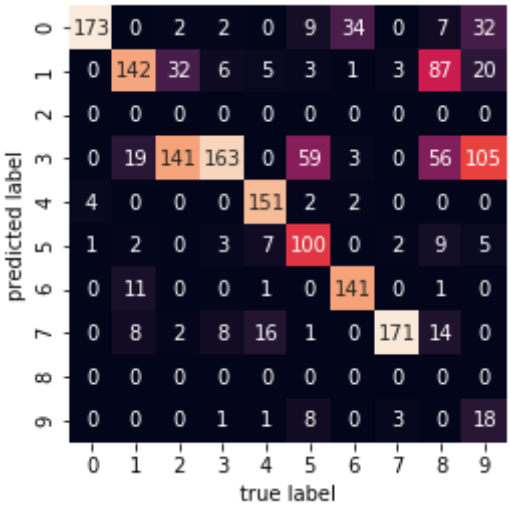
In [180]:
```
from scipy.stats import mode

labels = np.zeros_like(y_kmeans)
for i in range(10):
    mask = (y_kmeans == i)
    labels[mask] = mode(digits.target[mask])[0]
```

In [181]:
```
from sklearn.metrics import accuracy_score
accuracy_score(digits.target, labels)
```

Out[181]:
```
0.5893155258764607
```

In [183]:
```
import seaborn as sns;
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(digits.target, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=digits.target_names,
            yticklabels=digits.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```




As we might expect from the cluster centers we visualized before, the main point of confusion is between the eights, nines and ones. But this still shows that using k-means, we can essentially build a digit classifier without reference to any known labels!

In [ ]: