Unsupervised learning

Unsupervised learning is a method used to enable machines to classify objects without providing the machines any prior information about the objects. The main idea behind unsupervised learning is to expose the machines to large volumes of varied data and allow it to learn and infer from the data. However, the machines must first be programmed to learn from data.

1. Clustering

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group than those in other groups. The aim is to segregate groups with similar traits and assign them into clusters.

1.1. Hierarchical clustering

Hierarchical clustering, also known as hierarchical cluster analysis, is an algorithm that groups similar objects into groups called clusters. The endpoint is a set of clusters, where each cluster is distinct from each other cluster, and the objects within each cluster are broadly similar to each other.
This clustering technique is divided into two types:
1.Agglomerative
2.Divisive

1.1.1. Agglomerative Hierarchical clustering

Agglomerative Hierarchical clustering Technique: In this technique, initially each data point is considered as an individual cluster. At each iteration, the similar clusters merge with other clusters until one cluster or K clusters are formed. The

1.1.2. Divisive Hierarchical clustering

Divisive Hierarchical clustering Technique: Divisive Hierarchical clustering is exactly the opposite of the Agglomerative Hierarchical clustering. In Divisive Hierarchical clustering, we consider all the data points as a single cluster and in each iteration, we separate the data points from the cluster which are not similar. Each data point which is separated is considered as an individual cluster. In the end, we'll be left with n clusters.

Calculating the similarity between two clusters is important to merge or divide the clusters. There are different ways to find distance between the clusters. Following are some of the options to measure distance between two clusters:
1.Measure the distance between the closest points of two clusters.
2.Measure the distance between the farthest points of two clusters.
3.Measure the distance between the centroids of two clusters.
4.Measure the distance between all possible combination of points between the two clusters and take the mean.

The Hierarchical clustering Technique can be visualized using a Dendrogram. A Dendrogram is a tree-like diagram that records the sequences of merges or splits.

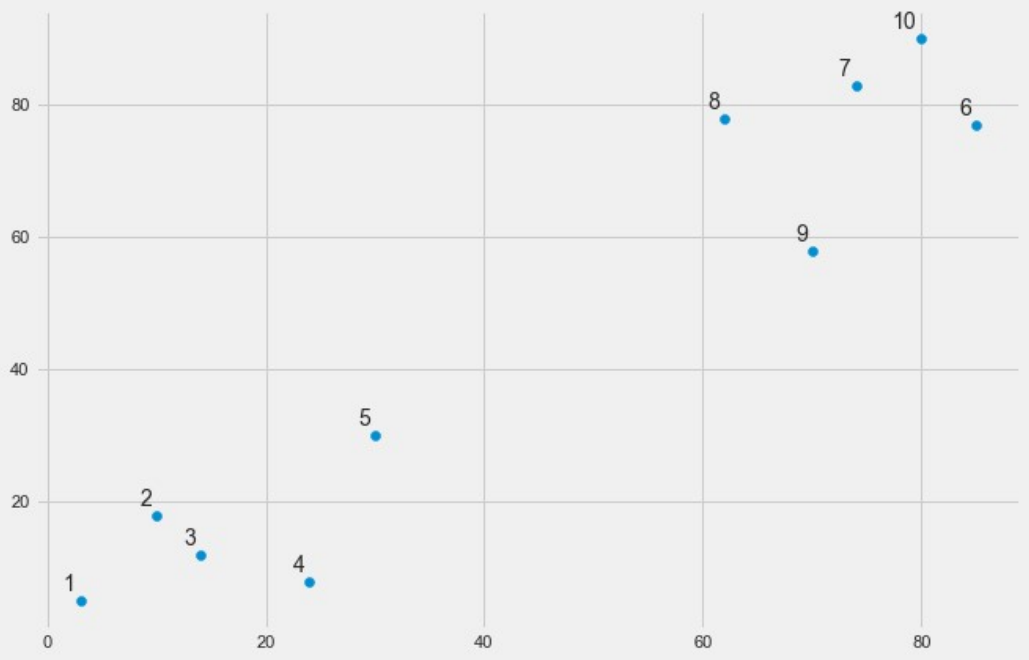
Once one large cluster is formed by the combination of small clusters, dendrograms of the cluster are used to actually split the cluster into multiple clusters of related data points. Let's see how it's actually done.

Suppose we have a collection of data points represented by a numpy array as follows:

In [50]:
```
import numpy as np
X = np.array([[3,5],
    [10,18],
    [14,12],
    [24,8],
    [30,30],
    [85,77],
    [74,83],
    [62,78],
    [70,58],
    [80,90],])
```

In [51]:

```
import matplotlib.pyplot as plt

labels = range(1, 11)
plt.figure(figsize=(10, 7))
plt.subplots_adjust(bottom=0.1)
plt.scatter(X[:,0],X[:,1], label='True Position')

for label, x, y in zip(labels, X[:, 0], X[:, 1]):
    plt.annotate(
        label,
        xy=(x, y), xytext=(-3, 3),
        textcoords='offset points', ha='right', va='bottom')
plt.show()
```



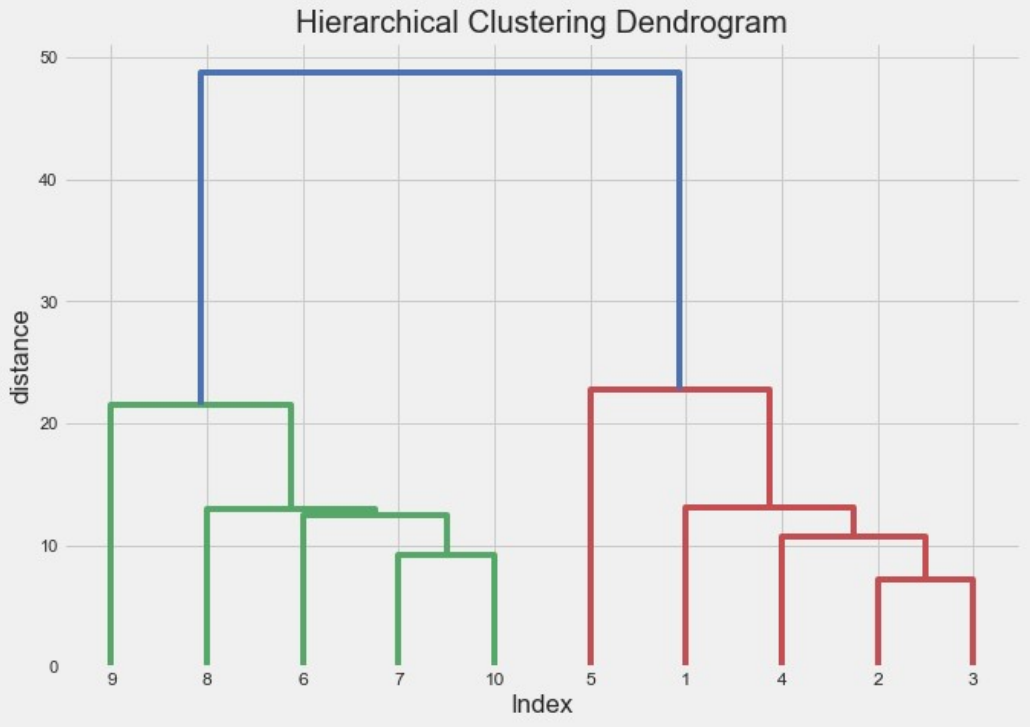
It can be seen from the naked eye that the data points form two clusters: first at the bottom left consisting of points 1-5 while second at the top right consisting of points 6-10.

However, in the real world, we may have thousands of data points in many more than 2 dimensions. In that case it would not be possible to spot clusters with the naked eye. This is why clustering algorithms have been developed.

Let's draw the dendrograms for our data points. We will use the scipy library for that purpose.

In [52]:

```
from scipy.cluster.hierarchy import dendrogram, linkage
from matplotlib import pyplot as plt

linked = linkage(X, 'single')

labelList = range(1, 11)

plt.figure(figsize=(10, 7))
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Index')
plt.ylabel('distance')
dendrogram(linked,
            orientation='top',
            labels=labelList,
            distance_sort='descending',
            show_leaf_counts=True)
plt.show()
```

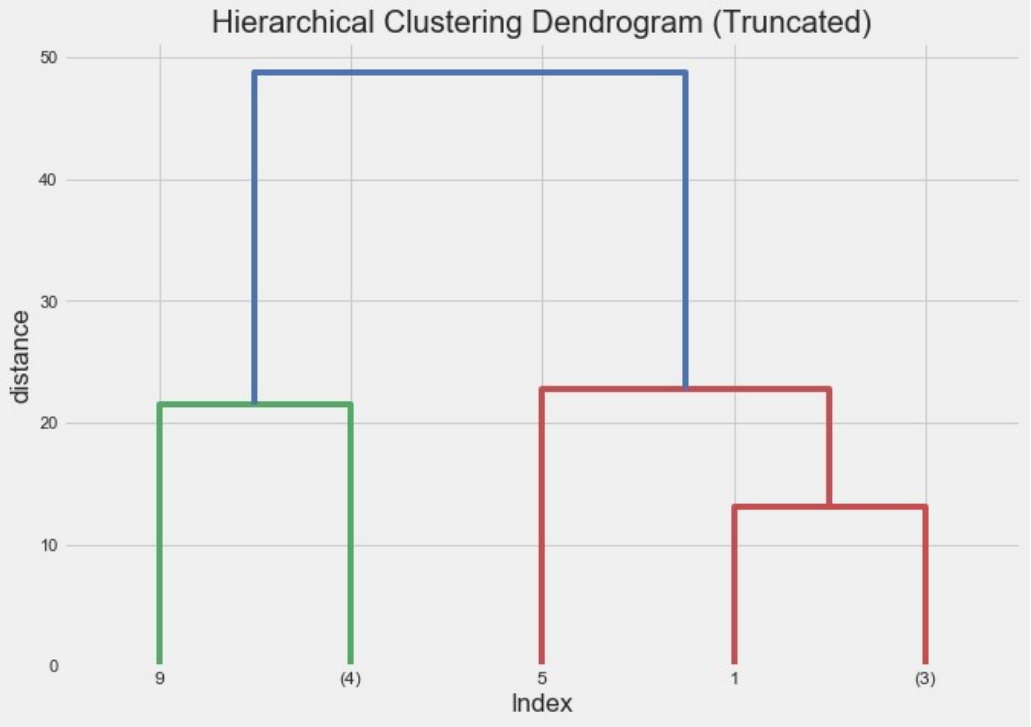


Dendrogram Truncation

We can truncate dendrogram according to the application of data, we can find last p clusters or can trucate it at any specific distance. Let's draw dendrogram of last 5 clusters of our data points.

In [53]:

```
plt.figure(figsize=(10, 7))
plt.title('Hierarchical Clustering Dendrogram (Truncated)')
plt.xlabel('Index')
plt.ylabel('distance')
dendrogram(linked,
            orientation='top',
            labels=labelList,
            distance_sort='descending',
            truncate_mode='lastp',  # show only the last p merged c
            p=5,  # show only the last p merged clusters
            show_leaf_counts=True)
plt.show()
```



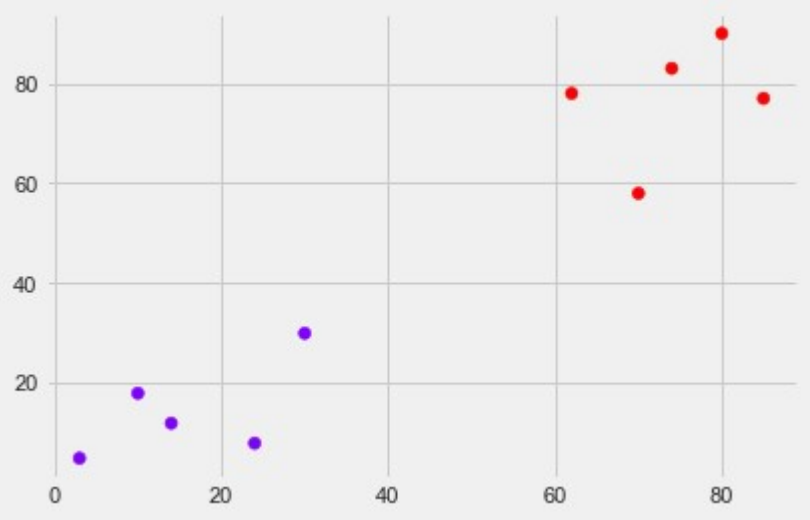
Above plot shows us last 5 clusters of our data points.

Now let's implement hierarchical clustering using Python's Scikit-Learn library.

In [54]:
```
from sklearn.cluster import AgglomerativeClustering

cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidear
cluster.fit_predict(X)
print(cluster.labels_)
plt.scatter(X[:,0],X[:,1], c=cluster.labels_, cmap='rainbow')
```

```
[0 0 0 0 0 1 1 1 1 1]
```

Out[54]:
```
<matplotlib.collections.PathCollection at 0x2828e4912e8>
```



We can see points in two clusters where the first five points clustered together and the last five points clustered together.

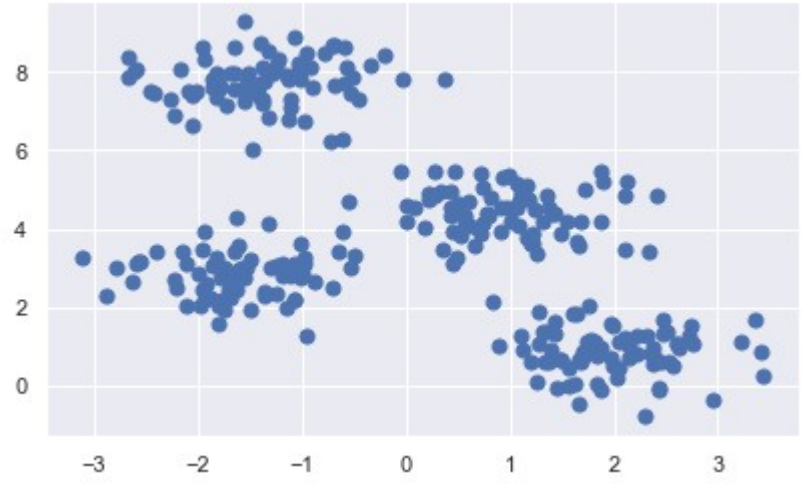
Advantages of Hierarchical clustering Technique:
1.No apriori information about the number of clusters required.
2.Easy to implement and gives best result in some cases.

Limitations of Hierarchical clustering Technique:
1.There is no mathematical objective for Hierarchical clustering.
2.All the approaches to calculate the similarity between clusters has its own disadvantages.
3.High space and time complexity for Hierarchical clustering. Hence this clustering algorithm cannot be used when we have huge data.

1.2. k-Means

In [55]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()  # for plot styling
import numpy as np
```

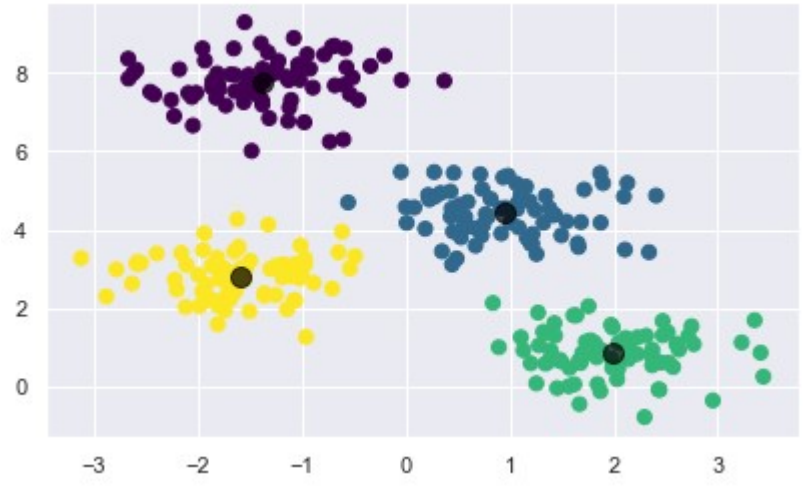
In [56]:
```
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=300, centers=4,
                       cluster_std=0.60, random_state=0)
plt.scatter(X[:, 0], X[:, 1], s=50);
```



By eye, it is relatively easy to pick out the four clusters. The k-means algorithm does this automatically, and in Scikit-Learn uses the typical estimator API:

In [57]:
```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
```

In [58]:
```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=100, alpha=0
```



How this algorithm finds these clusters so quickly! After all, the number of possible combinations of cluster assignments is exponential in the number of data points—an exhaustive search would be very, very costly, but such an exhaustive search is not necessary: instead, the typical approach to k-means involves an intuitive iterative approach known as expectation–maximization.

k-Means 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:

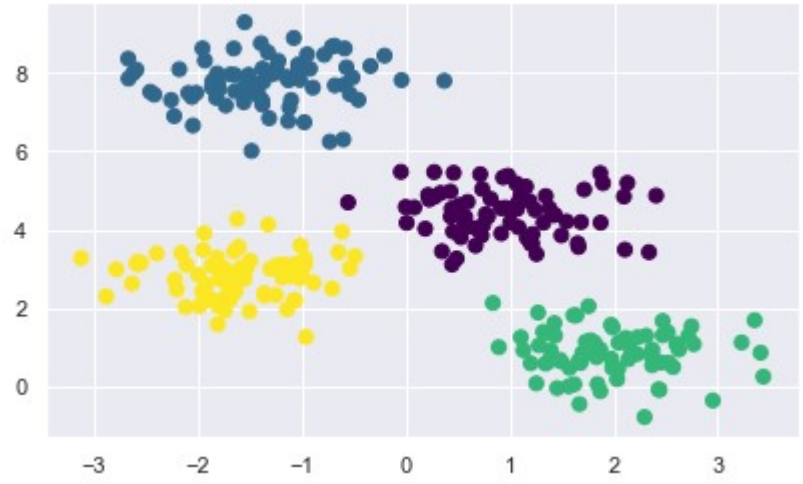
1.Guess some cluster centers

2.Repeat until converged

```
1.E-Step: assign points to the nearest cluster center
2.M-Step: set the cluster centers to the mean
```

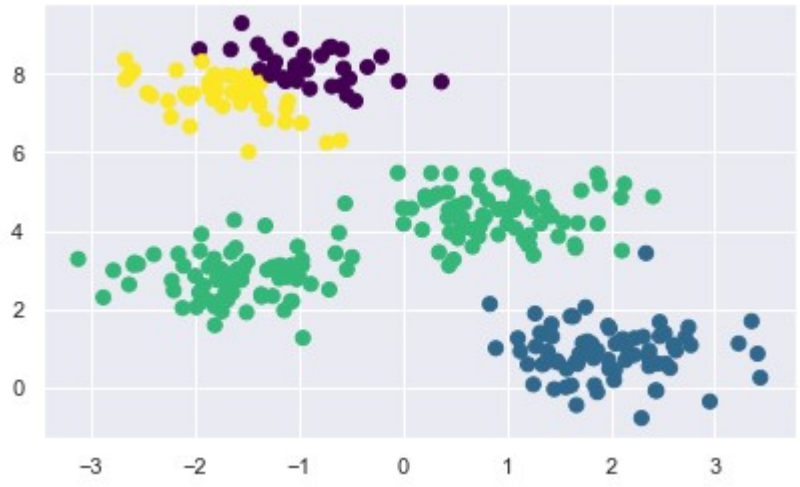
The k-Means algorithm is simple enough that we can write it in a few lines of code. The following is a very basic implementation:

In [59]:
```
from sklearn.metrics import pairwise_distances_argmin

def find_clusters(X, n_clusters, rseed=2):
    # 1. Randomly choose clusters
    rng = np.random.RandomState(rseed)
    i = rng.permutation(X.shape[0])[:n_clusters]
    centers = X[i]

    while True:
        # 2a. Assign labels based on closest center
        labels = pairwise_distances_argmin(X, centers)

        # 2b. Find new centers from means of points
        new_centers = np.array([X[labels == i].mean(0)
                                for i in range(n_clusters)])

        # 2c. Check for convergence
        if np.all(centers == new_centers):
            break
        centers = new_centers

    return centers, labels

centers, labels = find_clusters(X, 4)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```



There are a few issues to be aware of when using the expectation–maximization algorithm.

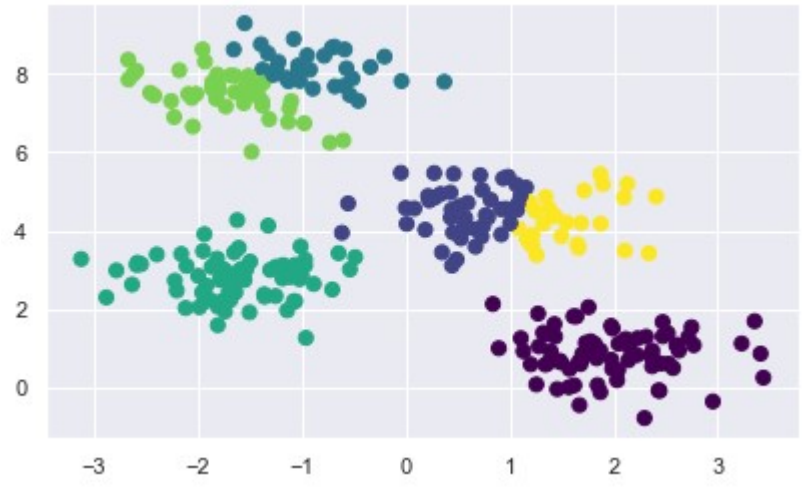
In [60]:
```
centers, labels = find_clusters(X, 4, rseed=3)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```



Here the E–M approach has converged, but has not converged to a globally optimal configuration. For this reason, it is common for the algorithm to be run for multiple starting guesses, as indeed Scikit-Learn does by default (set by the n_init parameter, which defaults to 10).

Another common challenge with k-means is that we must tell it how many clusters we expect: it cannot learn the number of clusters from the data. For example, if we ask the algorithm to identify six clusters, it will happily proceed and find the best six clusters:

In [61]:
```
labels = KMeans(6, random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```

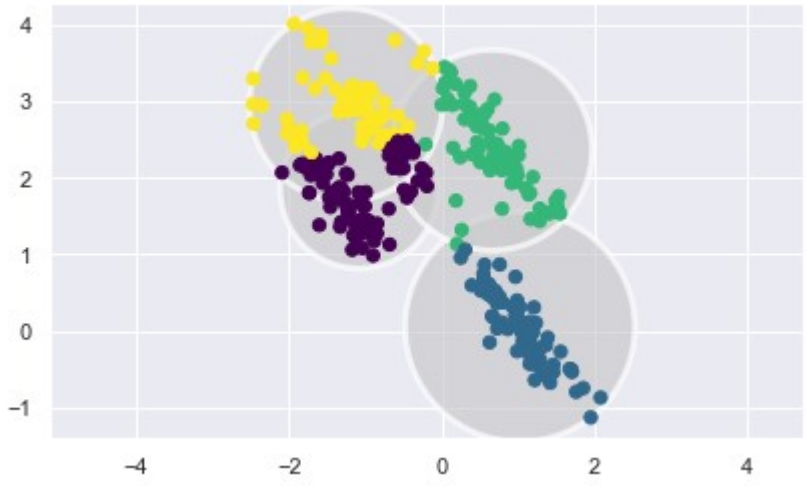


The fundamental model assumptions of k-means (points will be closer to their own cluster center than to others) means that the algorithm will often be ineffective if the clusters have complicated geometries.

An important observation for k-means is that these cluster models must be circular: k-means has no built-in way of accounting for oblong or elliptical clusters. So, for example, if we take the same data and transform it, the cluster assignments end up becoming muddled:

In [62]:

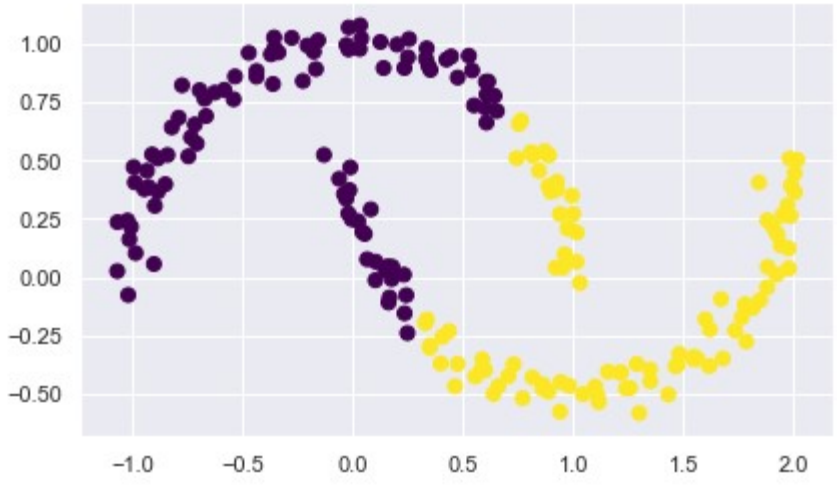
```
rng = np.random.RandomState(13)
X_stretched = np.dot(X, rng.randn(2, 2))

from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist

def plot_kmeans(kmeans, X, n_clusters=4, rseed=0, ax=None):
    labels = kmeans.fit_predict(X)

    # plot the input data
    ax = ax or plt.gca()
    ax.axis('equal')
    ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zo

    # plot the representation of the KMeans model
    centers = kmeans.cluster_centers_
    radii = [cdist(X[labels == i], [center]).max()
             for i, center in enumerate(centers)]
    for c, r in zip(centers, radii):
        ax.add_patch(plt.Circle(c, r, fc='#CCCCCC', lw=3, alpha=0.7

X, y_true = make_blobs(n_samples=400, centers=4,
                       cluster_std=0.60, random_state=0)
X = X[:, ::-1]

kmeans = KMeans(n_clusters=4, random_state=0)
plot_kmeans(kmeans, X_stretched)
```



In particular, the boundaries between k-means clusters will always be linear, which means that it will fail for more complicated boundaries. Consider the following data, along with the cluster labels found by the typical k-means approach:

In [63]:

```
from sklearn.datasets import make_moons
X, y = make_moons(200, noise=.05, random_state=0)
labels = KMeans(2, random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```



Advantages:

1.Fast, robust and easier to understand.

2.Relatively efficient: O(tknd), where n is number of objects, k is number of number of clusters, d is number of dimension of each object, and t is number of iterations. Normally, k, t, d << n.

3.Gives best result when data set are distinct or well separated from each other.

Disadvantages:

1.The learning algorithm requires apriori specification of the number of cluster centers.

2.If there are two highly overlapping data then k-means will not be able to resolve.

3.The learning algorithm is not invariant to non-linear transformations i.e. with different representation of data we get different results (data represented in form of cartesian co-ordinates and polar co-ordinates will give different results).

4.Euclidean distance measures can unequally weight underlying factors.

5.The learning algorithm provides the local optima of the squared error function.

6.Randomly choosing of the cluster center cannot lead us to the fruitful result. Pl. refer Fig.

7.Applicable only when mean is defined i.e. fails for categorical data.

8.Unable to handle noisy data and outliers.

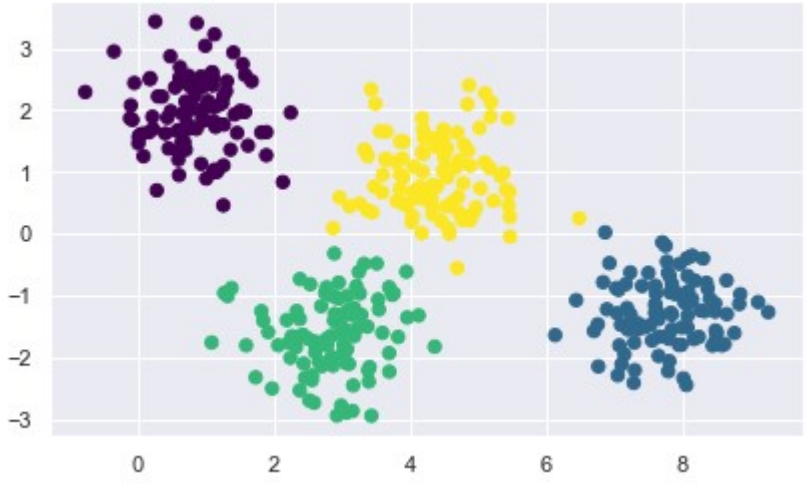
9.Algorithm fails for non-linear data set.

1.3. Gaussian Mixture Models

Gaussian Mixture models are used to find clusters in a dataset from which we know (or assume to know) the number of clusters enclosed in this dataset, but we do not know where these clusters are as well as how they are shaped. Finding these clusters is the task of GMM and since we don't have any information instead of the number of clusters, the GMM is an unsupervised approach. To accomplish that, we try to fit a mixture of gaussians to our dataset. That is, we try to find a number of gaussian distributions which can be used to describe the shape of our dataset. A critical point for the understanding is that these gaussian shaped clusters must not be circular shaped as for instance in the KNN approach but can have all shapes a multivariate Gaussian distribution can take. That is, a circle can only change in its diameter whilst a GMM model can (because of its covariance matrix) model all ellipsoid shapes as well.

In the simplest case, GMMs can be used for finding clusters in the same manner as k-means:

In [64]:
```
X, y_true = make_blobs(n_samples=400, centers=4,
                       cluster_std=0.60, random_state=0)
X = X[:, ::-1]
from sklearn.mixture import GaussianMixture as GMM
gmm = GMM(n_components=4).fit(X)
labels = gmm.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

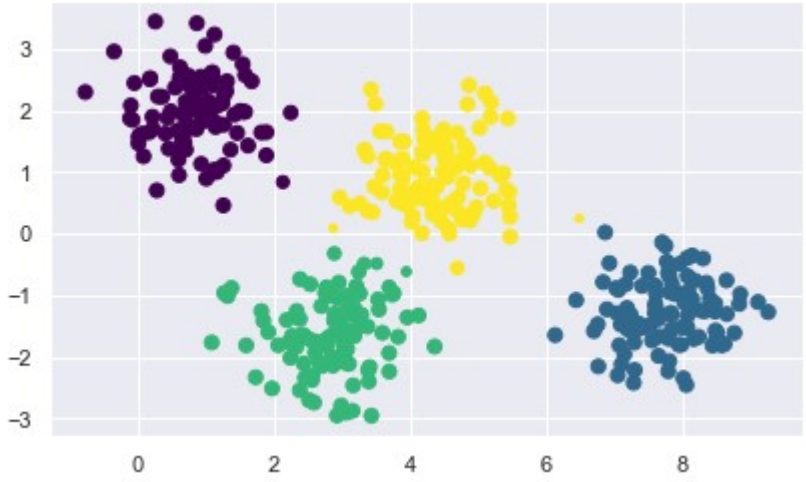


In [65]:
```
probs = gmm.predict_proba(X)
print(probs[:10].round(3))
```

```
[[0.    0.463 0.    0.537]
 [0.    0.    1.    0.   ]
 [0.    0.    1.    0.   ]
 [0.    0.    0.    1.   ]
 [0.    0.    1.    0.   ]
 [0.    1.    0.    0.   ]
 [1.    0.    0.    0.   ]
 [0.    0.    0.013 0.987]
 [0.    0.    1.    0.   ]
 [1.    0.    0.    0.   ]]
```

We can visualize this uncertainty by, for example, making the size of each point proportional to the certainty of its prediction; looking at the following figure, we can see that it is precisely the points at the boundaries between clusters that reflect this uncertainty of cluster assignment:

In [66]:
```
size = 50 * probs.max(1) ** 2  # square emphasizes differences
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=size);
```



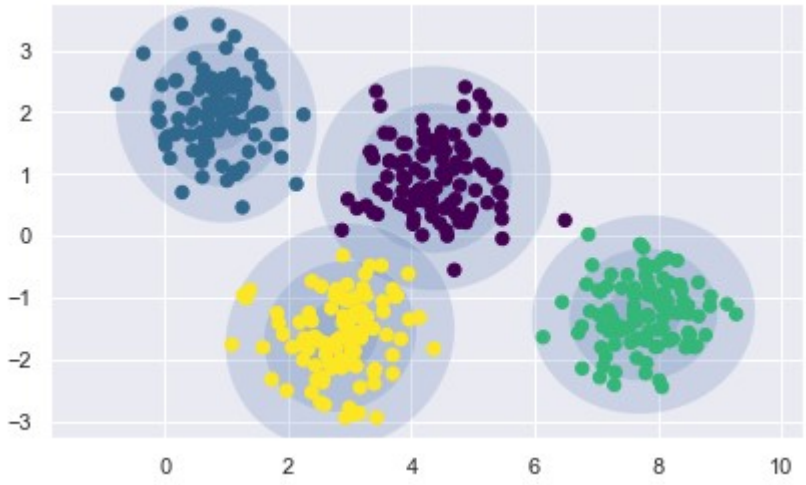
Under the hood, a Gaussian mixture model is very similar to k-means: it uses an expectation–maximization approach which qualitatively does the following:
1.Choose starting guesses for the location and shape
2.Repeat until converged:

In [67]:

```
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
    ax = ax or plt.gca()

    # Convert covariance to principal axes
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)

    # Draw the Ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))

def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis'
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')

    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weig
        draw_ellipse(pos, covar, alpha=w * w_factor)
```

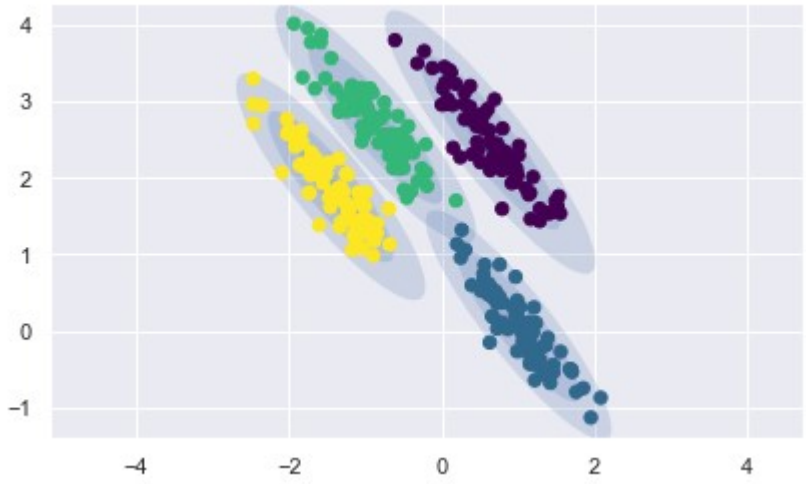
In [68]:

```
gmm = GMM(n_components=4, random_state=42)
plot_gmm(gmm, X)
```



Similarly, we can use the GMM approach to fit our stretched dataset; allowing for a full covariance the model will fit even very oblong, stretched-out clusters:

In [69]:
```
gmm = GMM(n_components=4, covariance_type='full', random_state=42)
plot_gmm(gmm, X_stretched)
```



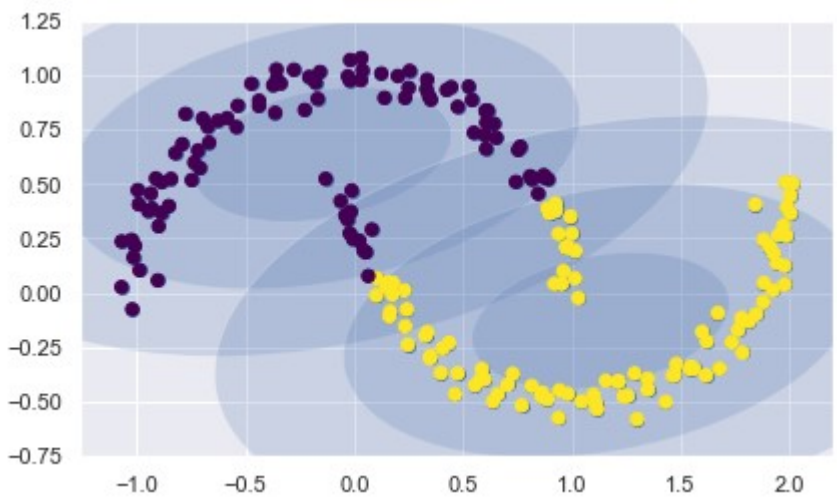
This makes clear that GMM addresses the two main practical issues with k-means encountered before.

Choosing the covariance type

If we look at the details of the preceding fits, we will see that the covariance_type option was set differently within each. This hyperparameter controls the degrees of freedom in the shape of each cluster; it is essential to set this carefully for any given problem. The default is covariance_type="diag", which means that the size of the cluster along each dimension can be set independently, with the resulting ellipse constrained to align with the axes. A slightly simpler and faster model is covariance_type="spherical", which constrains the shape of the cluster such that all dimensions are equal. The resulting clustering will have similar characteristics to that of k-means, though it is not entirely equivalent. A more complicated and computationally expensive model (especially as the number of dimensions grows) is to use covariance_type="full", which allows each cluster to be modeled as an ellipse with arbitrary orientation.

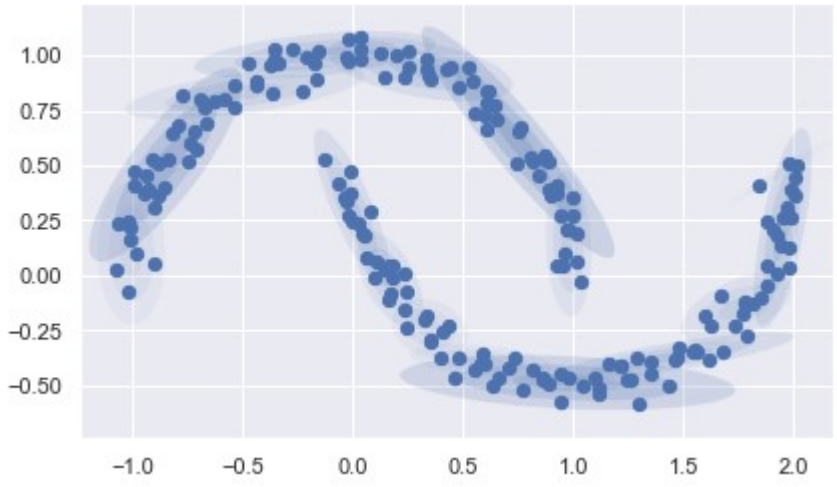
Though GMM is often categorized as a clustering algorithm, fundamentally it is an algorithm for density estimation. That is to say, the result of a GMM fit to some data is technically not a clustering model, but a generative probabilistic model describing the distribution of the data.

In [70]:
```
Xmoon, ymoon = make_moons(200, noise=.05, random_state=0)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
gmm2 = GMM(n_components=2, covariance_type='full', random_state=0)
plot_gmm(gmm2, Xmoon)
```



But if we instead use many more components and ignore the cluster labels, we find a fit that is much closer to the input data:

In [71]:
```
gmm16 = GMM(n_components=16, covariance_type='full', random_state=0
plot_gmm(gmm16, Xmoon, label=False)
```



Advantage:
1.Gives extremely useful result for the real world data set.

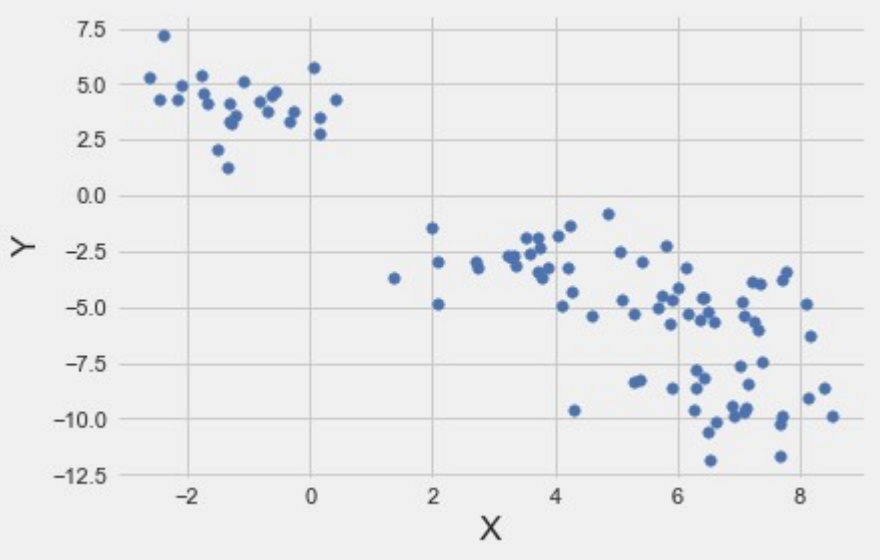
Disadvantage:
1.Algorithm is highly complex in nature.

Choosing optimal value of k clusters in k-means clustering

Elbow method is used to determine the optimal value of K to perform the K-Means Clustering Algorithm. The basic idea behind this method is that it plots the various values of cost with changing k. As the value of K increases, there will be fewer elements in the cluster. So average distortion will decrease. The lesser number of elements means closer to the centroid. So, the point where this distortion declines the most is the elbow point. Below is the Python implementation:

In [72]:

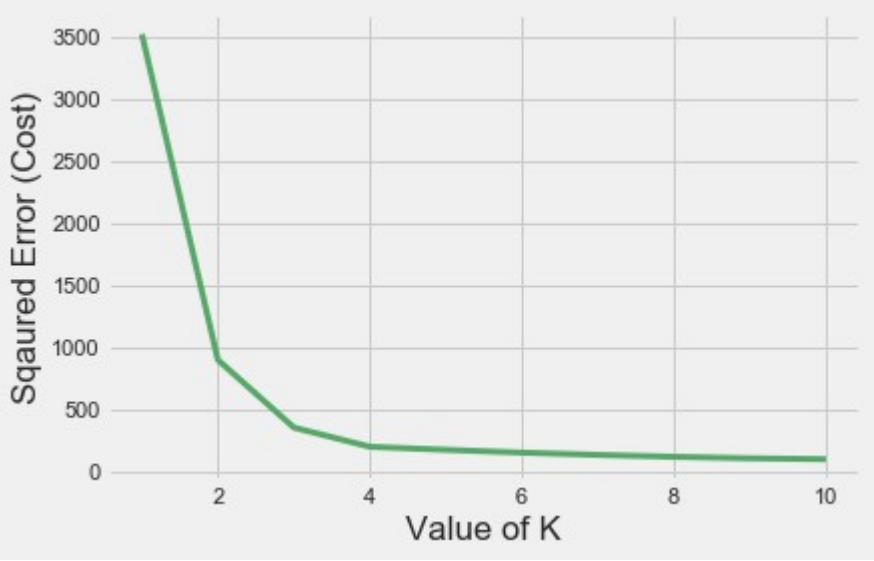
```
import matplotlib.pyplot as plt
from matplotlib import style
from sklearn.cluster import KMeans
from sklearn.datasets.samples_generator import make_blobs

style.use("fivethirtyeight")

# make_blobs() is used to generate sample points
# around c centers (randomly chosen)
X, y = make_blobs(n_samples = 100, centers = 4,
                  cluster_std = 1, n_features = 2)

plt.scatter(X[:, 0], X[:, 1], s = 30, color ='b')

# label the axes
plt.xlabel('X')
plt.ylabel('Y')

plt.show()
plt.clf() # clear the figure
```




```
<Figure size 432x288 with 0 Axes>
```

In [73]:

```
cost =[]
for i in range(1, 11):
    KM = KMeans(n_clusters = i, max_iter = 500)
    KM.fit(X)

    # calculates squared error
    # for the clustered points
    cost.append(KM.inertia_)

# plot the cost against K values
plt.plot(range(1, 11), cost, color ='g', linewidth ='3')
plt.xlabel("Value of K")
plt.ylabel("Sqaured Error (Cost)")
plt.show() # clear the plot

# the point of the elbow is the
# most optimal value for choosing k
```



In this case the optimal value for k would be 4. (Observable from the scattered points).

Questionnaire

What is Space and Time Complexity of Hierarchical clustering Technique?

Space complexity: The space required for the Hierarchical clustering Technique is very high when the number of data points are high as we need to store the similarity matrix in the RAM. The space complexity is the order of the square of n.

Space complexity = $O(n^2)$ where n is the number of data points.

Why Clustering?

Clustering is an important technique as it performs the determination of the intrinsic grouping among the unlabeled dataset. In clustering, there are no standard criteria. All of it depends on the user and the suitable criteria that satisfy their needs and requirements. For example, to find the homogeneous groups, one can find the representatives through data reduction and describe their suitable properties. One can also find unusual data objects for outlier detection.

What are some popular applications of clustering in machine learning?

Some of the popular applications of clustering in machine learning are –

1. Clustering Algorithm for identification of cancer cells

Cancerous Datasets can be identified using clustering algorithms. In a mix of data consisting of both cancerous and non-cancerous data, the clustering algorithms are able to learn the various features present in the data upon which they produce the resulting clusters. Through experimentation, we observe that the cancerous data set gives us accurate results when given a model of unsupervised non-linear clustering algorithm.

1. Clustering Algorithm in Search Engines

While searching for something particular on Google, we receive a mix of similar results that match to our original query. This is a result of clustering that groups similar objects in a single cluster and provides that to us. Based on the nearest similar object, the data is assigned to the single cluster providing a comprehensive set of results to the user.

1. Clustering Algorithm in Wireless Networks

Using the clustering algorithm on the wireless nodes, we are able to save energy utilized by the wireless sensors. There are various clustering-based algorithms in wireless networks to improve their energy consumption and optimize data transmission.

1. Clustering for Customer Segmentation

One of the most popular applications of clustering is in the field of customer segmentation. Based on the analysis of the user-base, companies are able to identify customers who would prove to be potential users for their product or services. Clustering allows them to segment customers into several clusters based on which they can adopt new strategies to appeal to their customer base.