**Clustering**

In contrast to *supervised* machine learning, *unsupervised* learning is used when there is no "ground truth" from which to train and validate label predictions. The most common form of unsupervised learning is *clustering*, which is simllar conceptually to *classification*, except that the the training data does not include known values for the class label to be predicted. Clustering works by separating the training cases based on similarities that can be determined from their feature values. Think of it this way; the numeric features of a given entity can be thought of as vector coordinates that define the entity's position in n-dimensional space. What a clustering model seeks to do is to identify groups, or *clusters*, of entities that are close to one another while being separated from other clusters.

For example, let's take a look at a dataset that contains measurements of different species of wheat seed.

**Citation**: The seeds dataset used in the this exercise was originally published by the Institute of Agrophysics of the Polish Academy of Sciences in Lublin, and can be downloaded from the UCI dataset repository (Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [[http://archive.ics.uci.edu/ml]](http://archive.ics.uci.edu/ml%5D). Irvine, CA: University of California, School of Information and Computer Science).

In [1]:



**import** pandas **as** pd

​

*# load the training dataset*

data **=** pd.read\_csv('data/seeds.csv')

​

*# Display a random sample of 10 observations (just the features)*

features **=** data[data.columns[0:6]]

features.sample(10)

Out[1]:

|  | **area** | **perimeter** | **compactness** | **kernel\_length** | **kernel\_width** | **asymmetry\_coefficient** |
| --- | --- | --- | --- | --- | --- | --- |
| **1** | 14.88 | 14.57 | 0.8811 | 5.554 | 3.333 | 1.018 |
| **13** | 13.78 | 14.06 | 0.8759 | 5.479 | 3.156 | 3.136 |
| **45** | 13.80 | 14.04 | 0.8794 | 5.376 | 3.155 | 1.560 |
| **61** | 11.23 | 12.63 | 0.8840 | 4.902 | 2.879 | 2.269 |
| **161** | 12.02 | 13.33 | 0.8503 | 5.350 | 2.810 | 4.271 |
| **52** | 14.49 | 14.61 | 0.8538 | 5.715 | 3.113 | 4.116 |
| **48** | 14.79 | 14.52 | 0.8819 | 5.545 | 3.291 | 2.704 |
| **187** | 10.91 | 12.80 | 0.8372 | 5.088 | 2.675 | 4.179 |
| **104** | 18.95 | 16.42 | 0.8829 | 6.248 | 3.755 | 3.368 |
| **109** | 18.55 | 16.22 | 0.8865 | 6.153 | 3.674 | 1.738 |

As you can see, the dataset contains six data points (or *features*) for each instance (*observation*) of a seed. So you could interpret these as coordinates that describe each instance's location in six-dimensional space.

Now, of course six-dimensional space is difficult to visualise in a three-dimensional world, or on a two-dimensional plot; so we'll take advantage of a mathematical technique called *Principal Component Analysis* (PCA) to analyze the relationships between the features and summarize each observation as coordinates for two principal components - in other words, we'll translate the six-dimensional feature values into two-dimensional coordinates.

In [2]:



**from** sklearn.preprocessing **import** MinMaxScaler

**from** sklearn.decomposition **import** PCA

​

*# Normalize the numeric features so they're on the same scale*

scaled\_features **=** MinMaxScaler().fit\_transform(features[data.columns[0:6]])

​

*# Get two principal components*

pca **=** PCA(n\_components**=**2).fit(scaled\_features)

features\_2d **=** pca.transform(scaled\_features)

features\_2d[0:10]

Out[2]:

array([[ 0.11883593, -0.09382469],

[ 0.0696878 , -0.31077233],

[-0.03499184, -0.37044705],

[-0.06582089, -0.36365235],

[ 0.32594892, -0.37695797],

[-0.02455447, -0.31060184],

[-0.00769646, -0.07594931],

[-0.05646955, -0.26696284],

[ 0.38196305, -0.05149471],

[ 0.35701044, -0.17697998]])

Now that we have the data points translated to two dimensions, we can visualize them in a plot:

In [4]:



**import** matplotlib.pyplot **as** plt

​

**%**matplotlib inline

​

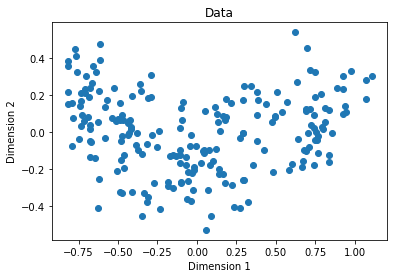
plt.scatter(features\_2d[:,0],features\_2d[:,1])

plt.xlabel('Dimension 1')

plt.ylabel('Dimension 2')

plt.title('Data')

plt.show()



Hopefully you can see at least two, arguably three, reasonably distinct groups of data points; but here lies one of the fundamental problems with clustering - without known class labels, how do you know how many clusters to separate your data into?

One way we can try to find out is to use a data sample to create a series of clustering models with an incrementing number of clusters, and measure how tightly the data points are grouped within each cluster. A metric often used to measure this tightness is the *within cluster sum of squares* (WCSS), with lower values meaning that the data points are closer. You can then plot the WCSS for each model.

In [5]:



*#importing the libraries*

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**from** sklearn.cluster **import** KMeans

**%**matplotlib inline

​

*# Create 10 models with 1 to 10 clusters*

wcss **=** []

**for** i **in** range(1, 11):

kmeans **=** KMeans(n\_clusters **=** i)

*# Fit the data points*

kmeans.fit(features.values)

*# Get the WCSS (inertia) value*

wcss.append(kmeans.inertia\_)

*#Plot the WCSS values onto a line graph*

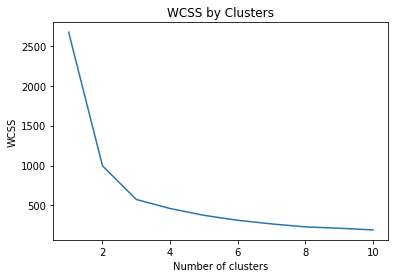
plt.plot(range(1, 11), wcss)

plt.title('WCSS by Clusters')

plt.xlabel('Number of clusters')

plt.ylabel('WCSS')

plt.show()



The plot shows a large reduction in WCSS (so greater *tightness*) as the number of clusters increases from one to two, and a further noticable reduction from two to three clusters. After that, the reduction is less pronounced, resulting in an "elbow" in the chart at around three clusters. This is a good indication that there are two to three reasonably well separated clusters of data points.

**K-Means Clustering**

The algorithm we used to create our test clusters is *K-Means*. This is a commonly used clustering algorithm that separates a dataset into *K* clusters of equal variance. The number of clusters, *K*, is user defined. The basic algorithm has the following steps:

1. A set of K centroids are randomly chosen.
2. Clusters are formed by assigning the data points to their closest centroid.
3. The means of each cluster is computed and the centroid is moved to the mean.
4. Steps 2 and 3 are repeated until a stopping criteria is met. Typically, the algorithm terminates when each new iteration results in negligable movement of centroids and the clusters become static.
5. When the clusters stop changing, the algorithm has *converged*, defining the locations of the clusters - note that the random starting point for the centroids means that re-running the algorithm could result in slightly different clusters, so training usually involves multiple iterations, reinitializing the centroids each time, and the model with the best WCSS is selected.

Let's try using K-Means on our seeds data with a K value of 3.

In [6]:



**from** sklearn.cluster **import** KMeans

​

*# Create a model based on 3 centroids*

model **=** KMeans(n\_clusters**=**3, init**=**'k-means++', n\_init**=**100, max\_iter**=**1000)

*# Fit to the data and predict the cluster assignments for each data point*

km\_clusters **=** model.fit\_predict(features.values)

*# View the cluster assignments*

km\_clusters

Out[6]:

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0,

0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0, 0,

0, 0, 0, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1,

0, 0, 0, 0, 1, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2], dtype=int32)

Let's see those cluster assignments with the two-dimensional data points.

In [7]:



**def** plot\_clusters(samples, clusters):

col\_dic **=** {0:'blue',1:'green',2:'orange'}

mrk\_dic **=** {0:'\*',1:'x',2:'+'}

colors **=** [col\_dic[x] **for** x **in** clusters]

markers **=** [mrk\_dic[x] **for** x **in** clusters]

**for** sample **in** range(len(clusters)):

plt.scatter(samples[sample][0], samples[sample][1], color **=** colors[sample], marker**=**markers[sample], s**=**100)

plt.xlabel('Dimension 1')

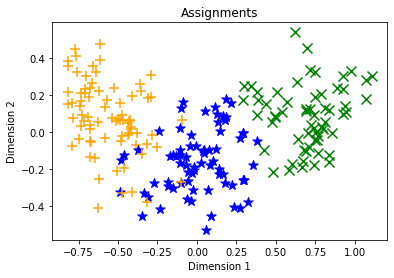
plt.ylabel('Dimension 2')

plt.title('Assignments')

plt.show()

​

plot\_clusters(features\_2d, km\_clusters)



Hopefully, the the data has been separated into three distinct clusters.

So what's the practical use of clustering? In some cases, you may have data that you need to group into distict clusters without knowing how many clusters there are or what they indicate. For example a marketing organization might want to separate customers into distinct segments, and then investigate how those segments exhibit different purchasing behaviors.

Sometimes, clustering is used as an initial step towards creating a classification model. You start by identifying distinct groups of data points, and then assign class labels to those clusters. You can then use this labelled data to train a classification model.

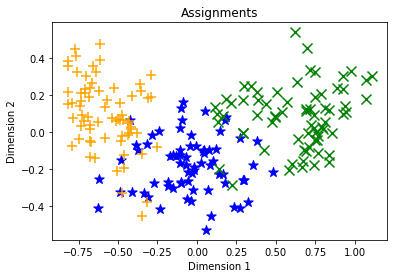
In the case of the seeds data, the different species of seed are already known and encoded as 0 (*Kama*), 1 (*Rosa*), or 2 (*Canadian*), so we can use these identifiers to compare the species classifications to the clusters identified by our unsupervised algorithm

In [8]:



seed\_species **=** data[data.columns[7]]

plot\_clusters(features\_2d, seed\_species.values)



There may be some differences between the cluster assignments and class labels, but the K-Means model should have done a reasonable job of clustering the observations so that seeds of the same species are generally in the same cluster.

**Hierarchical Clustering**

Hierarchical clustering methods make fewer distributional assumptions when compared to K-means methods. However, K-means methods are generally more scalable, sometimes very much so.

Hierarchical clustering creates clusters by either a *divisive* method or *agglomerative* method. The divisive method is a "top down" approach starting with the entire dataset and then finding partitions in a stepwise manner. Agglomerative clustering is a "bottom up\*\* approach. In this lab you will work with agglomerative clustering which roughly works as follows:

1. The linkage distances between each of the data points is computed.
2. Points are clustered pairwise with their nearest neighbor.
3. Linkage distances between the clusters are computed.
4. Clusters are combined pairwise into larger clusters.
5. Steps 3 and 4 are repeated until all data points are in a single cluster.

The linkage function can be computed in a number of ways:

* Ward linkage measures the increase in variance for the clusters being linked,
* Average linkage uses the mean pairwise distance between the members of the two clusters,
* Complete or Maximal linkage uses the maximum distance between the members of the two clusters.

Several different distance metrics are used to compute linkage functions:

* Euclidian or l2 distance is the most widely used. This metric is only choice for the Ward linkage method.
* Manhattan or l1 distance is robust to outliers and has other interesting properties.
* Cosine similarity, is the dot product between the location vectors divided by the magnitudes of the vectors. Notice that this metric is a measure of similarity, whereas the other two metrics are measures of difference. Similarity can be quite useful when working with data such as images or text documents.

**Agglomerative Clustering**

Let's see an example of clustering the seeds data using an agglomerative clustering algorithm.

In [9]:



**from** sklearn.cluster **import** AgglomerativeClustering

​

agg\_model **=** AgglomerativeClustering(n\_clusters**=**3)

agg\_clusters **=** agg\_model.fit\_predict(features.values)

agg\_clusters

Out[9]:

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0,

0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0,

0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1,

0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,

2, 0, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2])

So what do the agglomerative cluster assignments look like?

In [10]:



**import** matplotlib.pyplot **as** plt

​

**%**matplotlib inline

​

**def** plot\_clusters(samples, clusters):

col\_dic **=** {0:'blue',1:'green',2:'orange'}

mrk\_dic **=** {0:'\*',1:'x',2:'+'}

colors **=** [col\_dic[x] **for** x **in** clusters]

markers **=** [mrk\_dic[x] **for** x **in** clusters]

**for** sample **in** range(len(clusters)):

plt.scatter(samples[sample][0], samples[sample][1], color **=** colors[sample], marker**=**markers[sample], s**=**100)

plt.xlabel('Dimension 1')

plt.ylabel('Dimension 2')

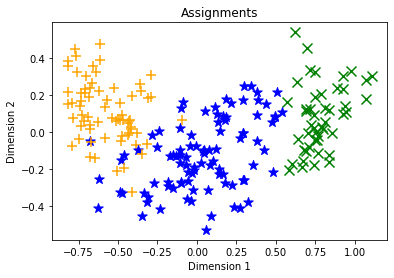
plt.title('Assignments')

plt.show()

​

plot\_clusters(features\_2d, agg\_clusters)

​



**Further Reading**

To learn more about clustering with scikit-learn, see the [scikit-learn documentation](https://scikit-learn.org/stable/modules/clustering.html).

**Further Reading**

To learn more about the Python packages you explored in this notebook, see the following documentation:

* [NumPy](https://numpy.org/doc/stable/)
* [Pandas](https://pandas.pydata.org/pandas-docs/stable/)
* [Matplotlib](https://matplotlib.org/contents.html)

**Challenge: Cluster Unlabelled Data**

Now that you've seen how to create a clustering model, why not try for yourself? You'll find a clustering challenge in the [/challenges/04 - Clustering Challenge.ipynb](https://newcomputeinst.eastus2.instances.azureml.ms/notebooks/Users/ml-basics/challenges/04%20-%20Clustering%20Challenge.ipynb) notebook!

**Note**: The time to complete this optional challenge is not included in the estimated time for this exercise - you can spend as little or as much time on it as you like!