Using KMeans to Cluster Data

- Clustering is a very useful technique. Often, we need to divide and conquer when taking actions.
- Consider a list of potential customers for a business. A business might need to group customers into cohorts, and then departmentalize responsibilities for these cohorts.
- Clustering can help facilitate the clustering process.
- KMeans is probably one of the most well-known clustering algorithms and, in a larger sense, one of the most well-known unsupervised learning techniques.

Getting ready

First, let's walk through some simple clustering, then we'll talk about how KMeans works. Also, since we'll be doing some plotting, import matplotlib as shown.

```
>>> from sklearn.datasets import make_blobs
>>> blobs, classes = make_blobs(500, centers=3)
>>
>>> import matplotlib.pyplot as plt
```

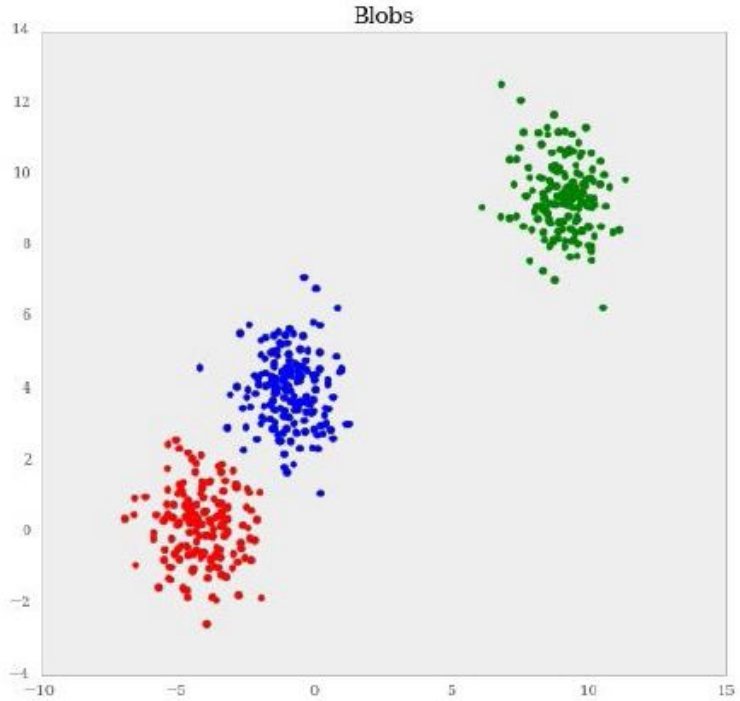
Implementation

- We are going to walk through a simple example that clusters blobs of fake data.
- Then we'll talk a little bit about how KMeans works to find the optimal number of blobs.

- Looking at our blobs, we can see that there are three distinct clusters:

```
>>> f, ax = plt.subplots(figsize=(7.5, 7.5))
>>> ax.scatter(blobs[:, 0], blobs[:, 1], color=rgb[classes])
>>> rgb = np.array(['r', 'g', 'b'])
>>> ax.set_title("Blobs")
```

The output is as follows:



Now we can use KMeans to find the centers of these clusters. In the first example, we'll work on the basis that there are three centers:

```
>>> from sklearn.cluster import KMeans
```

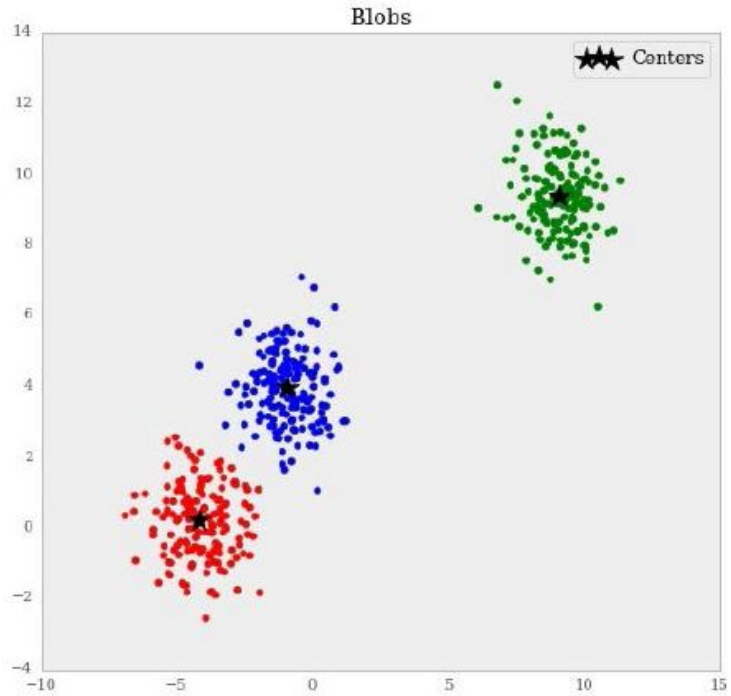
```
>>> kmean = KMeans(n_clusters=3)
>>> kmean.fit(blobs)
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=3,
n_init=10, n_jobs=1, precompute_distances=True,
random_state=None, tol=0.0001, verbose=0)
>>>
>>> kmean.cluster_centers_
array([[ 0.47819567, 1.80819197],

[ 0.08627847, 8.24102715],
[ 5.2026125 , 7.86881767]])
>>> f, ax = plt.subplots(figsize=(7.5, 7.5))
>>> ax.scatter(blobs[:, 0], blobs[:, 1], color=rgb[classes])
>>> ax.scatter(kmean.cluster_centers_[:, 0],
kmean.cluster_centers_[:, 1], marker='*', s=250,
color='black', label='Centers')
>>> ax.set_title("Blobs")
>>> ax.legend(loc='best')
```

The following screenshot shows the output: Other attributes are useful too. For instance, the `labels_` attribute will produce the expected label for each point:

```
>>> kmean.labels_[:5]
array([1, 1, 2, 2, 1], dtype=int32)
```

We can check whether `kmean.labels_` is the same as classes, but because KMeans has no knowledge of the classes going in, it cannot assign the sample index values to both classes:




```
>>> classes[:5]
array([0, 0, 2, 2, 0])
```

Feel free to swap 1 and 0 in classes to see if it matches up with `labels_`. The transform function is quite useful in the sense that it will output the distance between each point and centroid:

```
>>> kmean.transform(blobs)[:5]
array([[ 6.47297373, 1.39043536, 6.4936008 ],
[ 6.78947843, 1.51914705, 3.67659072],
[ 7.24414567, 5.42840092, 0.76940367],
[ 8.56306214, 5.78156881, 0.89062961],
[ 7.32149254, 0.89737788, 5.1224679]])
```

Implementation

KMeans is actually a very simple algorithm that works to minimize the within-cluster sum of square distances from the mean. We'll be minimizing the sum of squares yet again!

It does this by first setting a pre-specified number of clusters, K, and then alternating between the following:

- Assigning each observation to the nearest cluster
- Updating each centroid by calculating the mean of each observation assigned to this cluster

This happens until some specified criterion is met.