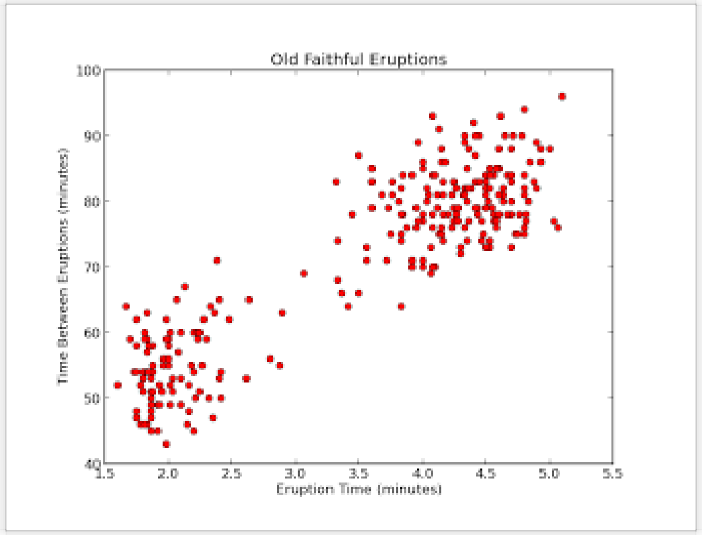
MAY
17

Clustering using scikit-learn

The Old Faithful data set is a set of historical observations showing the waiting time before an eruption and the length of the eruption. In the last post we looked into it a little and I'm going to continue looking into it in this post. To refresh (or initialize) everyone's memory of the data set, here is the scatterplot of the data again:



The data appears to be split into two, possibly more, separate populations and in this post we'll examine a simple clustering technique to automatically classify observations as being in one cluster or another. There are many different clustering techniques out there and the simplest of these is K-means clustering. K-means clustering doesn't assume any underlying probability model, but instead just acts on the data. Given a target number, k, of clusters to find, it will locate the centers of each of those k clusters and the boundaries between them. It does this using the following algorithm:

1. Start with a randomly selected set of k centroids (the supposed centers of the k clusters)

squared Euclidean distance: \sum_{j=1}^p (x_{ij} - x_{i'j})^2 where p is the number of dimensions)
3. Re-calculate the centroids of each cluster by minimizing the squared Euclidean distance to each observation in the cluster
4. Repeat 2. and 3. until the members of the clusters (and hence the positions of the centroids) no longer change.

This is a kind of iterative descent algorithm, where you repeatedly find the minimum until it converges. A potential issue with this kind of algorithm is that it is not guaranteed to find the most optimal cluster arrangement, if you pick the wrong starting points. One method for overcoming this is to run the algorithm a number of times with different randomly selected starting points, and then pick the solution that has the lowest total squared Euclidean distance. This approach is used in the scikit-learn package, defaulting to 10 separate repetitions.

Scikit-learn uses numpy arrays, so make sure you format your data accordingly, and is extremely easy to use. To generate clusters using K-means, you only need do the following (remember all complete code is available in my GitHub repository):

```
from sklearn import cluster

k = 2
kmeans = cluster.KMeans(n_clusters=k)
kmeans.fit(data)
```

And to get the locations of the centroids and the label of the owning cluster for each observation in the data set:

```
labels = kmeans.labels_
centroids = kmeans.cluster_centers_
```

Using these, we can now plot the chosen clusters and their calculated centroids:

```
from matplotlib import pyplot
import numpy as np

for i in range(k):
    # select only data observations with cluster label == i
    ds = data[np.where(labels==i)]
    # plot the data observations
    pyplot.plot(ds[:,0],ds[:,1],'o')
    # plot the centroids
    lines = pyplot.plot(centroids[i,0],centroids[i,1],'kx')
    # make the centroid x's bigger
    pyplot.setp(lines,ms=15.0)
    pyplot.setp(lines,mew=2.0)
pyplot.show()
```

This gives the following cluster assignments for 2 clusters and for 6 clusters:

The scikit-learn package also has a function that allows you to get the centroids and labels directly:

```
from sklearn import cluster
centroids,labels,inertia = cluster.k_means(data,n_clusters=k)
```

Using the KMeans object directly, however, will allow us to use them to make predictions of which cluster a new observation belongs to, which we can do now. To do this, we need to split the data into a training set and a test set, we will then do the K-means clustering on the training set and use the test set to make predictions. A simple way to do a random split of the data is:

```
import numpy as np
import numpy.random as npr

def split_data(data,train_split=0.8):
    data = np.array(data)
    num_train = data.shape[0] * train_split
    npr.shuffle(data)

    return (data[:num_train],data[num_train:])
```

```
training_data,test_data = split_data(data)
```

The predictions are then simply calculated:

```
labels = kmeans.predict(test_data)
```

Plotting the predictions, as before, for 2 clusters and 6 clusters:

One of the features of K-means clustering is that the partitions between clusters is a hard threshold; observations on one side are in one cluster and on the the other are in another cluster. This is often not a desirable feature as it can lead to misclassifcation near the boundaries.

Another shortcoming of K-means clustering is that changing the number of clusters can change the boundaries in arbitrary ways -- in the Old Faithful example, moving from 2 clusters to 6 clusters did preserve the original k=2 boundary, but it's not guaranteed in every example.

In the next post, we'll look at some different methods for cluster analysis.

Posted 17th May 2013 by Barney Govan

Labels: clustering, k-means, numpy, python, scikit-learn

+1 Tweet Like

1 **View comments**