Anomaly Detection in Network Traffic with K-means clustering

We can categorize machine learning algorithms into two main groups: **supervised learning** and **unsupervised learning**. With supervised learning algorithms, in order to predict unknown values for new data, we have to know the target value for many previously-seen examples. In contrast, unsupervised learning algorithms explore the data which has no target attribute to find some intrinsic structures in them.

Clustering is a technique for finding similar groups in data, called **clusters**. Clustering is often called an unsupervised learning task as no class values denoting an a priori grouping of the data instances are given.

In this notebook, we will use K-means, a very well known clustering algorithm to detect anomaly network connections based on statistics about each of them. For a thorough overview of K-means clustering, from a research perspective, have a look at this wonderful tutorial (http://theory.stanford.edu/~sergei/slides/kdd10-thclust.pdf).

Goals

We expect students to:

- Learn (or revise) and understand the K-means algorithm
- Implement a simple K-means algorithm
- Use K-means to detect anomalies network connection data

Steps

1. In section 1, we will have an overview about K-means then implement a simple version of it.
2. In section 2, we build models with and without categorical features.
3. Finally, in the last section, using our models, we will detect unsual connections.

1. K-means

1.1. Introduction

Clustering is a typical and well-known type of unsupervised learning. Clustering algorithms try to find natural groupings in data. Similar data points (according to some notion of similarity) are considered in the same group. We call these groups **clusters**.

K-Means clustering is a simple and widely-used clustering algorithm. Given value of k, it tries to build k clusters from samples in the dataset. Therefore, k is an hyperparameter of the model. The right value of k is not easy to determine, as it highly depends on the data set and the way that data is featurized.

To measue the similarity between any two data points, K-means requires the definition of a distance funtion between data points. What is a distance? It is a value that indicates how close two data points are in their space. In particular, when data points lie in a d-dimensional space, the Euclidean distance is a good choice of a distance function, and is supported by MLLIB.

In K-means, a cluster is a group of points, with a representative entity called a centroid. A centroid is also a point in the data space: the center of all the points that make up the cluster. It's defined to be the arithmetic mean of the points. In general, when working with K-means, each data sample is represented in a d-dimensional numeric vector, for which it is easier to define an appropriate distance function. As a consequence, in some applications, the original data must be transformed into a different representation, to fit the requirements of K-means.

1.2. How does it work ?

Given k, the K-means algorithm works as follows:

1. Randomly choose k data points (seeds) to be the initial centroids
2. Assign each data point to the **closest centroid**
3. Re-compute (update) the centroids using the current cluster memberships
4. If a convergence criterion is not met, go to step 2

We can also terminate the algorithm when it reaches an iteration budget, which yields an approximate result. From the pseudo-code of the algorithm, we can see that K-means clustering results can be sensitive to the order in which data samples in the data set are explored. A sensible practice would be to run the analysis several times, randomizing objects order; then, average the cluster centres of those runs and input the centres as initial ones for one final run of the analysis.

1.3. Illustrative example

One of the best ways to study an algorithm is trying implement it. In this section, we will go step by step to implement a simple K-means algorithm.

Question 1

Question 1.1

Complete the below function to calculate an euclidean distance between any two points in d-dimensional data space

In [1]:

```
import math
import numpy as np

# calculate distance between two d-dimensional points
def euclidean_distance(p1, p2):
    return np.linalg.norm(np.array(p2)-np.array(p1))

# test our function
assert (round(euclidean_distance([1,2,3] , [10,18,12]), 2) == 20.45), "Function's wrong"
```

Question 1.2

Given a data point and the current set of centroids, complete the function below to find the index of the closest centroid for that data point.

In [3]:

```
#################################################################
def find_closest_centroid(datapoint, centroids):
    # find the index of the closest centroid of the given data point.
    distances=[euclidean_distance(datapoint, centroids[i]) for i in range(len(centroids))]
    return np.argmin(distances)

assert(find_closest_centroid( [1,1,1], [ [2,1,2], [1,2,1], [3,1,2] ] ) == 1), "Function's wrong"
```

Question 1.3

Write a function to randomize `k` initial centroids.

In [4]:

```
np.random.seed(22324)
import random
# randomize initial centroids
def randomize_centroids(data, k):
    centroids = random.sample(list(data),k)

    return centroids

assert(len(
    randomize_centroids(
        np.array([
            np.array([2,1,2]),
            np.array([1,2,1]),
            np.array([3,1,2])
             ]),
        2)) == 2), "Wrong function"
```

Question 1.4

Write function `check_converge` to check the stop creteria of the algorithm.

In [5]:

```
MAX_ITERATIONS = 100

# return True if clusters have converged , otherwise, return False
def check_converge(centroids, old_centroids, num_iterations, threshold=0.00001):
    # if it reaches an iteration budget
    if (num_iterations>=MAX_ITERATIONS):
        return True
    # check if the centroids don't move (or very slightly)

    l=np.array(list(map(lambda x,y : euclidean_distance(x, y) , centroids, old_centroids)))#################
#

    if (sum(l*l)<=threshold):
        return True
    else :
        return False
```

Question 1.5

Write function `update_centroid` to update the new positions for the current centroids based on the position of their members.

In [6]:

```
# centroids: a list of centers
# cluster: a list of k elements. Each element i-th is a list of data points that are assigned to center i-th
def update_centroids(centroids, cluster):
    centroids=[[]for i in range(len(cluster))]
    for i in range(len(centroids)):
        centroids[i]=np.mean(cluster[i],axis=0)
    return centroids
```

Question 1.6

Complete the K-means algorithm scheleton below, with the functions you wrote above.

In [7]:

```
# data : set of data points
# k : number of clusters
# centroids: initial list of centroids
def kmeans(data, k=2, centroids=None):

    # randomize the centroids if they are not given
    if not centroids:
        centroids = randomize_centroids(data, k)

    old_centroids = centroids[:]
    print(centroids)
    iterations = 0
    while True:
        iterations += 1

        # init empty clusters
        clusters = [[] for i in range(k)]

        # assign each data point to the closest centroid
        for datapoint in data:
            # find the closest center of each data point
            centroid_idx = find_closest_centroid(datapoint, centroids)

            # assign datapoint to the closest cluster

            clusters[centroid_idx].append(datapoint)

        # keep the current position of centroids before changing them
        old_centroids = centroids

        # update centroids
        centroids = update_centroids(centroids, clusters)

        # if the stop criteria are met, stop the algorithm
        if check_converge(centroids, old_centroids, iterations, threshold=0.00001):
            break

    return centroids
```

Next, we will test our algorithm on Fisher's Iris dataset (http://en.wikipedia.org/wiki/Iris_flower_data_set), and plot the resulting clusters in 3D.

Question 1.7

The code below can be used to test your algorithm with three different datasets: `Iris`, `Moon` and `Blob`. Run your algorithm to cluster datapoints in these datasets, plot the results and discuss about them. Do you think that our algorithm works well? Why?

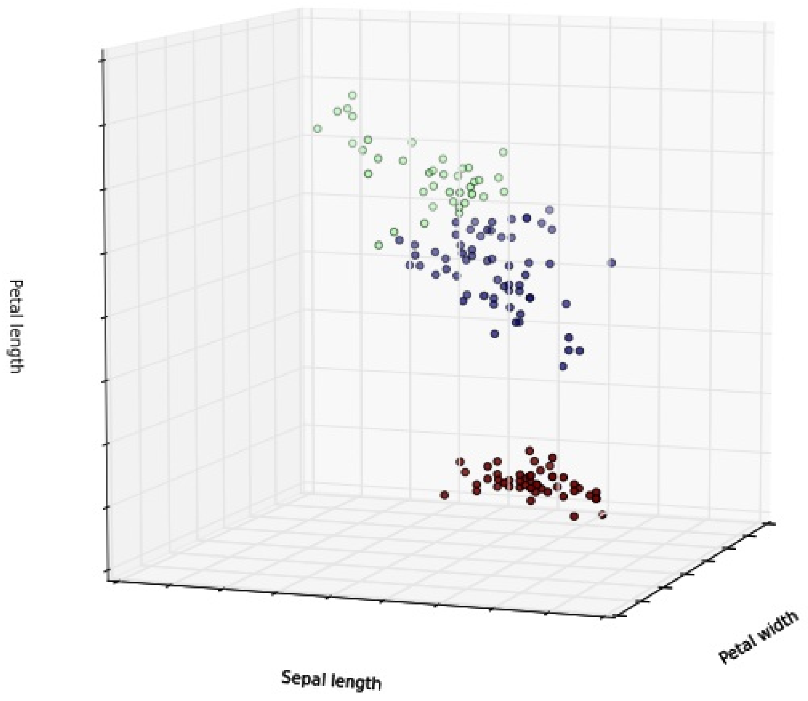
In [14]:

```
# the sourcecode in this cell is inspired from
# https://gist.github.com/bbarrilleaux/9841297

%matplotlib inline

from sklearn import datasets, cluster
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# load data
iris = datasets.load_iris()
X_iris = iris.data
y_iris = iris.target
# do the clustering
centers = kmeans(X_iris, k=3)
labels = [find_closest_centroid(p, centers) for p in X_iris]

#plot the clusters in color
fig = plt.figure(1, figsize=(8, 8))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, 1, 1], elev=8, azim=200)
plt.cla()
ax.scatter(X_iris[:, 3], X_iris[:, 0], X_iris[:, 2], c=labels)



ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])
ax.set_xlabel('Petal width')
ax.set_ylabel('Sepal length')
ax.set_zlabel('Petal length')

plt.show()
```

```
/opt/conda/lib/python3.5/site-packages/sklearn/utils/fixes.py:64: DeprecationWarning: inspect.g
etargspec() is deprecated, use inspect.signature() instead
  if 'order' in inspect.getargspec(np.copy)[0]:
```

```
[array([ 6.1,  3. ,  4.6,  1.4]), array([ 6.5,  3.2,  5.1,  2. ]), array([ 4.9,  3. ,  1.4,  0.
2])]
```

```
/opt/conda/lib/python3.5/site-packages/matplotlib/collections.py:590: FutureWarning: elementwis
e comparison failed; returning scalar instead, but in the future will perform elementwise compa
rison
  if self._edgecolors == str('face'):
```

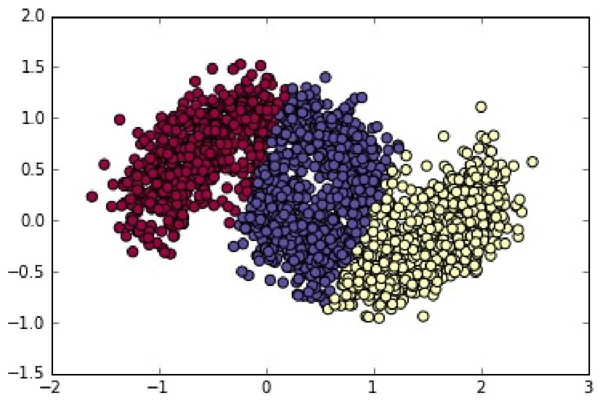


In [42]:

```
# moon
np.random.seed(0)
X, y = datasets.make_moons(2000, noise=0.2)
# do the clustering
centers = kmeans(X, k=3)
labels = [find_closest_centroid(p, centers) for p in X]

plt.scatter(X[:,0], X[:,1], s=40, c=labels, cmap=plt.cm.Spectral)
plt.show()
```

```
[array([-0.52970811,  1.03789462]), array([ 0.95632652, -0.47697944]), array([-0.41665124,  0.9
3656486])]
```

```
/opt/conda/lib/python3.5/site-packages/matplotlib/collections.py:590: FutureWarning: elementwis
e comparison failed; returning scalar instead, but in the future will perform elementwise compa
rison
  if self._edgecolors == str('face'):
```

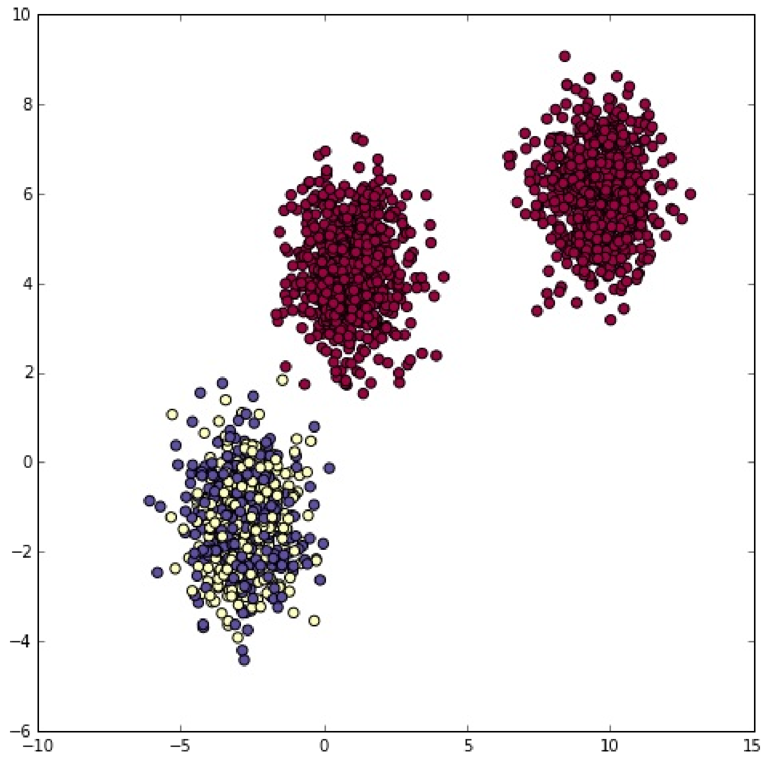


In [41]:

```
# blob

plt.figure(figsize=(8,8))
np.random.seed(0)
X, y = datasets.make_blobs(n_samples=2000, centers=3, n_features=20, random_state=0)

centers = kmeans(X, k=3)
labels = [find_closest_centroid(p, centers) for p in X]

plt.clf()
plt.scatter(X[:,0], X[:,1], s=40, c=labels, cmap=plt.cm.Spectral)
plt.show()
```

```
[array([-0.40939547,  3.28705541,  2.37582556,  1.32713101, -2.31266666,
        2.72722046,  0.17310966,  6.44085557,  9.28105566, -3.40718742,
        7.05272547, -0.60090668,  0.99377731,  8.38958392, -9.25225733,
       -8.58653141, -9.63453956,  7.24457062,  3.86540116,  7.15112958]), array([ -2.78502653,
 -1.95998638,   4.58578548,  -8.20386126,
         3.3921596 ,   3.57893578,  -5.51097491,  -6.33486844,
        -4.70327555,  -2.81178195,   0.64044206,   0.87536424,
        10.10457788,  -9.05970007,  -7.18805105,  -7.61818086,
         4.21995858,  -5.55828827,  -2.5267618 ,  -5.78203741]), array([ -3.49890053,  -2.52892
001,   4.88601327,  -7.67341798,
         2.63017046,   3.06601292,  -4.79702356,  -6.86592853,
        -3.78280187,  -1.73463966,   1.46043041,  -0.05157875,
        10.19428729,  -9.64382457,  -5.38440607,  -7.61405494,
         3.88919266,  -4.96516548,  -1.38968217,  -4.90380562])]
```

```
/opt/conda/lib/python3.5/site-packages/matplotlib/collections.py:590: FutureWarning: elementwis
e comparison failed; returning scalar instead, but in the future will perform elementwise compa
rison
  if self._edgecolors == str('face'):
```

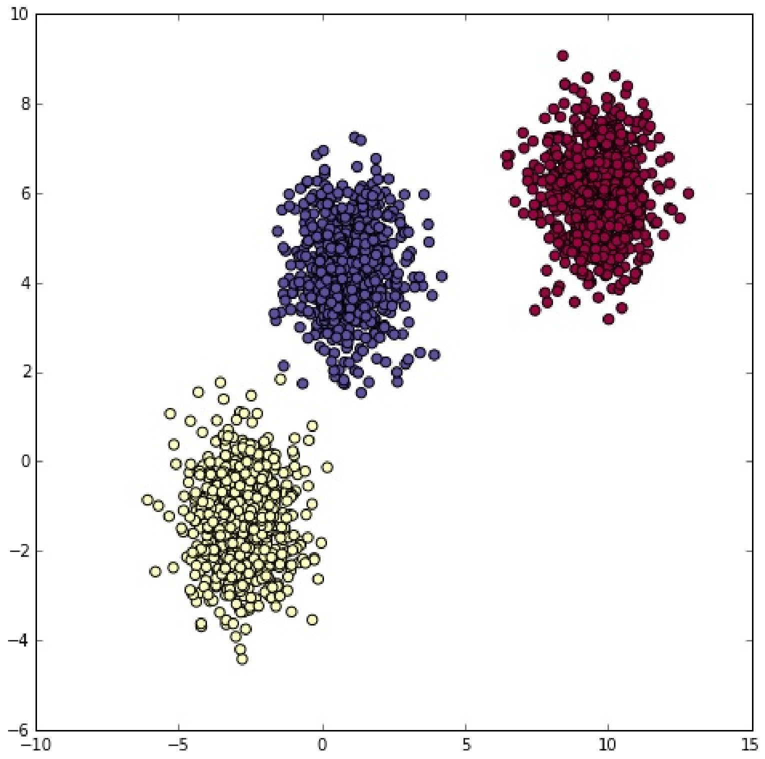


In [43]:

```
# blob

plt.figure(figsize=(8,8))
np.random.seed(0)
X, y = datasets.make_blobs(n_samples=2000, centers=3, n_features=20, random_state=0)

centers = kmeans(X, k=3)
labels = [find_closest_centroid(p, centers) for p in X]

plt.clf()
plt.scatter(X[:,0], X[:,1], s=40, c=labels, cmap=plt.cm.Spectral)
plt.show()
```

```
[array([ 10.00761707,   6.01930526,  -1.11634982,   4.28236738,
        -8.02701693,   2.41083294,  -7.57569529,   9.52816937,
         1.19730331,  -2.15299063,  -4.45948163,   3.76481544,
        -3.40109776,   2.40811816, -10.43708544,   1.00861637,
         1.29093205,   2.85727974,   9.53619288,   3.25982019]), array([  7.6482671 ,   5.91360
265,   0.25866661,   4.92423107,
        -6.04240048,   2.28342972,  -6.70161999,  10.12590081,
         1.17514875,  -1.65611303,  -4.43701256,   5.43877656,
        -1.158341  ,   1.51575263,  -8.50010765,   1.87974675,
         2.54511684,   2.30081117,   9.63015198,   3.90909495]), array([ 1.50635301,  3.8791525
1,  1.45452095,  2.27465081, -1.11289305,
        3.94564376, -2.07311716,  8.09257076,  8.56356522, -2.48317228,
        5.21161033,  0.43780026,  1.16658525,  7.14381634, -9.74327139,
       -7.82659031, -9.46172511,  5.84072591,  5.03485648,  7.86304359])]
```

```
/opt/conda/lib/python3.5/site-packages/matplotlib/collections.py:590: FutureWarning: elementwis
e comparison failed; returning scalar instead, but in the future will perform elementwise compa
rison
  if self._edgecolors == str('face'):
```



We can see that in the last plot, the clustering is quite good since we can distinguish three clusters, and the points in each cluster are close to each other. Regarding the first plot, the points from different clusters are more close to each other but still we can distinguish three separated clusters and the clustering seems to be good. But for the second plot, the clustering is bad and eventhough we can see three cluster, they points are not clustered correctly. We can say that this is due to the random initialization of centroids and that in this the clustering algorithms outcome is very correlated to initial centroids.

That's enough about K-means for now. In the next section, we will apply MMLIB's K-means on Spark to deal with a large data in the real usecase.

2. Usecase: Network Intrusion

Some attacks attempt to flood a computer with network traffic. In some other cases, attacks attempt to exploit flaws in networking software in order to gain unauthorized access to a computer. Detecting an exploit in an incredibly large haystack of network requests is not easy.

Some exploit behaviors follow known patterns such as scanning every port in a short of time, sending a burst of request to a port... However, the biggest threat may be the one that has never been detected and classified yet. Part of detecting potential network intrusions is detecting anomalies. These are connections that aren't known to be attacks, but, do not resemble connections that have been observed in the past.

In this notebook, K-means is used to detect anomalous network connections based on statistics about each of them.

2.1. Data

The data comes from KDD Cup 1999 (http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html). The dataset is about 708MB and contains about 4.9M connections. For each connection, the data set contains information like the number of bytes sent, login attempts, TCP errors, and so on. Each connection is one line of CSV-formatted data, containing 38 features: back, buffer_overflow, ftp_write, guess_passwd, imap, ipsweep, land, loadmodule, multihop, neptune, nmap, normal, perl, phf, pod, portsweep, rootkit, satan, smurf, spy, teardrop, warezclient, warezmaster. For more details about each features, please follow this link (http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html).

Many features take on the value 0 or 1, indicating the presence or absence of a behavior such as `su_attempted` in the 15th column. Some features are counts, like `num_file_creations` in the 17th columns. Some others are the number of sent and received bytes.

2.2. Clustering without using categorical features

First, we need to import some packages that are used in this notebook.

In [27]:

```
import os
import sys
import re
from pyspark import SparkContext
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.types import *
from pyspark.sql import Row
from pyspark.sql.functions import *
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import pyspark.sql.functions as func
import matplotlib.patches as mpatches
from pyspark.mllib.clustering import KMeans, KMeansModel

input_path = "/datasets/k-means/kddcup.data"
raw_data = sc.textFile(input_path, 12)
```

2.2.1. Loading data

There are two types of features: numerical features and categorical features. Currently, to get familiar with the data and the problem, we only use numerical features. In our data, we also have pre-defined groups for each connection, which we can use later as our "ground truth" for verifying our results.

Note 1: we don't use the labels in the training phase !!!

Note 2: in general, since clustering is un-supervised, you don't have access to ground truth. For this reason, several metrics to judge the quality of clustering have been devised. For a short overview of such metrics, follow this link (https://en.wikipedia.org/wiki/Cluster_analysis#Internal_evaluation). Note that computing such metrics, that is trying to assess the quality of your clustering results, is as computationally intensive as computing the clustering itself!

In [6]:

```
raw_data.take(5)
```

Out[6]:

```
['0,tcp,http,SF,215,45076,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.0
0,0,0,0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00,normal.',
 '0,tcp,http,SF,162,4528,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00
,1,1,1.00,0.00,1.00,0.00,0.00,0.00,0.00,0.00,normal.',
 '0,tcp,http,SF,236,1228,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00
,2,2,1.00,0.00,0.50,0.00,0.00,0.00,0.00,0.00,normal.',
 '0,tcp,http,SF,233,2032,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00
,3,3,1.00,0.00,0.33,0.00,0.00,0.00,0.00,0.00,normal.',
 '0,tcp,http,SF,239,486,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,3,3,0.00,0.00,0.00,0.00,1.00,0.00,0.00,
4,4,1.00,0.00,0.25,0.00,0.00,0.00,0.00,0.00,normal.']
```

Question 2

Write function `parseLine` to construct a tuple of `(label, vector)` for each connection, extract the data that contains only the data points (without label), then print the number of connections.

Where,

- `label` is the pre-defined label of each connection
- `vector` is a numpy array that contains values of all features, but the label and the categorial features at index `1,2,3` of each connection. Each `vector` is a data point.

In [28]:

```
def parseLine(line):
    cols = line.split(',')
    # label is the last column
    label = cols[len(cols)-1]

    # vector is every column, except the label
    vector = cols[0:len(cols)-1]

    # delete values of columns that have index 1->3 (categorical features)
    del vector[1]
    del vector[1]
    del vector[1]

    # convert each value from string to float
    vector = np.array(list(float(x) for x in vector))

    return (label, vector)

labelsAndData = raw_data.map(parseLine)

# we only need the data, not the label
data = labelsAndData.map(lambda x : x[1]).cache()

# number of connections
n = data.count()
```

In [10]:

```
print(n)
```

```
4898431
```

In [5]:

```
print(data.take(5))
```

```
[array([  0.00000000e+00,   2.15000000e+02,   4.50760000e+04,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   1.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   1.00000000e+00,   1.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   1.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00]), array([  0.00000000e+00,   1.62000000e+02,   4.528
00000e+03,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   1.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   2.00000000e+00,   2.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   1.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   1.00000000e+00,   1.00000000e+00,
         1.00000000e+00,   0.00000000e+00,   1.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00]), array([  0.00000000e+00,   2.36000000e+02,   1.228
00000e+03,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   1.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   1.00000000e+00,   1.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   1.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   2.00000000e+00,   2.00000000e+00,
         1.00000000e+00,   0.00000000e+00,   5.00000000e-01,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00]), array([  0.00000000e+00,   2.33000000e+02,   2.032
00000e+03,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   1.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   2.00000000e+00,   2.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   1.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   3.00000000e+00,   3.00000000e+00,
         1.00000000e+00,   0.00000000e+00,   3.30000000e-01,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00]), array([  0.00000000e+00,   2.39000000e+02,   4.860
00000e+02,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   1.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   3.00000000e+00,   3.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   1.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   4.00000000e+00,   4.00000000e+00,
         1.00000000e+00,   0.00000000e+00,   2.50000000e-01,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00])]
```

Question 3

Using K-means algorithm of MLLIB, cluster the connections into two groups then plot the result. Why two groups? In this case, we are just warming up, we're testing things around, so "two groups" has no particular meaning.

You can use the following parameters:

- `maxIterations=10`
- `runs=10`
- `initializationMode="random"`

Discuss the result from your figure.

In [11]:

```
import pandas as pd
```

In [87]:

```
from pyspark.mllib.clustering import KMeans

model = KMeans.train(data, 2, maxIterations=10,runs=10, initializationMode="random")
```

```
/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")
```

In [88]:

```
print("Final centers: " + str(model.clusterCenters))
```

```
Final centers: [array([  1.19636396e+02,   9.12169640e+01,   2.68038098e+03,
         1.41942699e-05,   1.47975263e-03,   1.97705902e-05,
         2.85162882e-02,   7.95892989e-05,   3.00098802e-01,
         1.90051135e-02,   1.68810424e-04,   8.51656192e-05,
         2.06835859e-02,   1.76617272e-03,   1.11526406e-05,
         1.88023382e-03,   0.00000000e+00,   1.01387642e-06,
         2.07388422e-03,   1.09917536e+02,   1.12936551e+01,
         4.41858070e-01,   4.42019181e-01,   1.43131773e-01,
         1.43227214e-01,   4.79341584e-01,   5.07767409e-02,
         4.94657987e-02,   2.08163235e+02,   9.98043644e+01,
         4.13329053e-01,   7.20208281e-02,   7.28242643e-02,
         1.52164094e-02,   4.41816126e-01,   4.41572147e-01,
         1.43271840e-01,   1.43050977e-01]), array([  2.74821895e-01,   3.01005405e+03,   2.380
33751e+01,
         0.00000000e+00,   8.85226762e-05,   0.00000000e+00,
         1.59716782e-03,   0.00000000e+00,   3.79669998e-02,
         7.28005020e-04,   3.41786394e-07,   4.10143673e-06,
         7.71070106e-03,   7.99438377e-04,   1.16890947e-04,
         4.41929808e-04,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   4.86709972e+02,   4.86726821e+02,
         5.26282690e-05,   5.56291536e-05,   4.19952943e-05,
         8.64890471e-05,   9.99257291e-01,   1.22471293e-03,
         1.39640352e-02,   2.49713700e+02,   2.49495913e+02,
         9.83206062e-01,   2.85938498e-03,   9.63889044e-01,
         5.63154606e-04,   2.83265728e-04,   1.04067121e-04,
         3.87397789e-04,   8.69641302e-05])]
```

In [89]:

```
clusterLabelCountt = data.map(lambda x: (model.predict(x), 1)).reduceByKey(lambda a,b : a+b).collect()

for item in clusterLabelCountt :
    print(item)
```

```
(0, 1972627)
(1, 2925804)
```

In [90]:

```
ourpdf=data.sample(False, 0.02)
```

In [91]:

```
ourpdf.count()
```

Out[91]:

```
98503
```

In [92]:

```
pdf =pd.DataFrame(data=ourpdf.collect())
```

In [93]:

```
from sklearn.decomposition import PCA

pca = PCA(n_components=3)
pca.fit(pdf)
```

```
/opt/conda/lib/python3.5/site-packages/sklearn/base.py:175: DeprecationWarning: inspect.getargs
pec() is deprecated, use inspect.signature() instead
  args, varargs, kw, default = inspect.getargspec(init)
```

Out[93]:

```
PCA(copy=True, n_components=3, whiten=False)
```

In [94]:

```
exd2 = pca.transform(pdf)
```

In [95]:

```
pdfex2d = pd.DataFrame(exd2)
pdfex2d.index = pdf.index
pdfex2d.columns = ['PC1','PC2','PC3']
pdfex2d.head()
```

Out[95]:

	PC1	PC2	PC3
0	-614.800036	4850.919672	22.530296
1	-711.538246	1433.890518	22.362211
2	-379.681880	8779.406433	24.868959
3	-764.603116	-336.473002	24.639170
4	-630.308479	3506.328384	27.055790

In [98]:

```
np.max(X[:, 2])
```

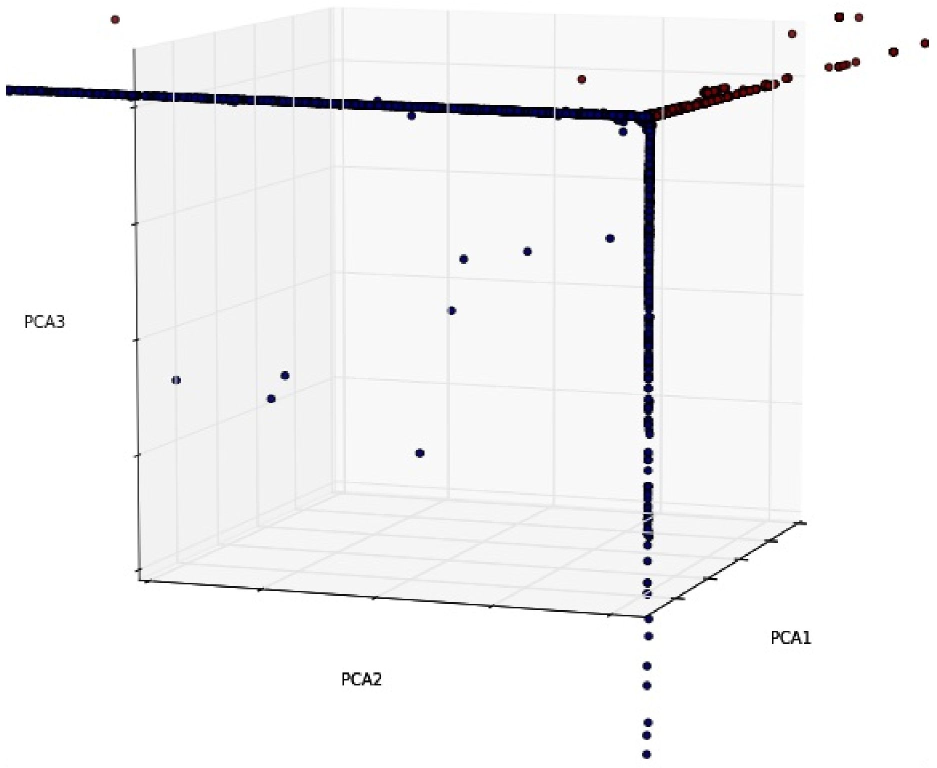
Out[98]:

```
1464.8174731460674
```

In [97]:

```
%matplotlib inline

from sklearn import datasets, cluster
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# load data

X=np.array(pdfex2d)

# do the clustering
#centers = model.clusterCenters
labels = [model.predict(p) for p in ourpdf.collect()]

#plot the clusters in color
fig = plt.figure(1, figsize=(8, 8))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, 1, 1], elev=8, azim=200)
plt.cla()
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=labels)



ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])
ax.set_xlim(-4000,20000)
ax.set_ylim(-1000,20000)
ax.set_zlim(-20000,2000)
ax.set_xlabel('PCA1')
ax.set_ylabel('PCA2')
ax.set_zlabel('PCA3')

plt.show()
```

```
/opt/conda/lib/python3.5/site-packages/matplotlib/collections.py:590: FutureWarning: elementwis
e comparison failed; returning scalar instead, but in the future will perform elementwise compa
rison
  if self._edgecolors == str('face'):
```



Using the PCA (principal component analysis) we got the graph above. We used this method because we wanted to represent the clustering in 3D while we have 38 features. This method seems to be efficient because it takes the first principal component so as to have the largest possible variance and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components. So this can potentially help as cluster the data based on the most relevant features regarding anomaly detection.

2.2.3. Evaluating model

Question 4

One of the simplest method to evaluate our result is calculate the Within Set Sum of Squared Errors (WSSSE), or simply, 'Sum of Squared Errors'. An error of a data point is defined as it's distance to the closest cluster center.

In [12]:

```
from operator import add

# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(clusters, point):
    closest_center = clusters.centers[clusters.predict(point)]
    return np.sum([x**2 for x in (point - closest_center)])
```

In [14]:

```
from operator import add

# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(clusters, point):
    closest_center = clusters.centers[clusters.predict(point)]
    return np.sum([x**2 for x in (point - closest_center)])

WSSSE = data.map(lambda x : error(model,x)).reduce(add)
print("Within Set Sum of Squared Error = " + str(WSSSE))
```

```
Within Set Sum of Squared Error = 6.37937321457e+18
```

Question 5

This is a good opportunity to use the given labels to get an intuitive sense of what went into these two clusters, by counting the labels within each cluster. Complete the following code that uses the model to assign each data point to a cluster, and counts occurrences of cluster and label pairs. What do you think about the result ?

In [48]:

```
model = KMeans.train(data, 2, maxIterations=10,runs=10, initializationMode="random")
```

```
/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")
```

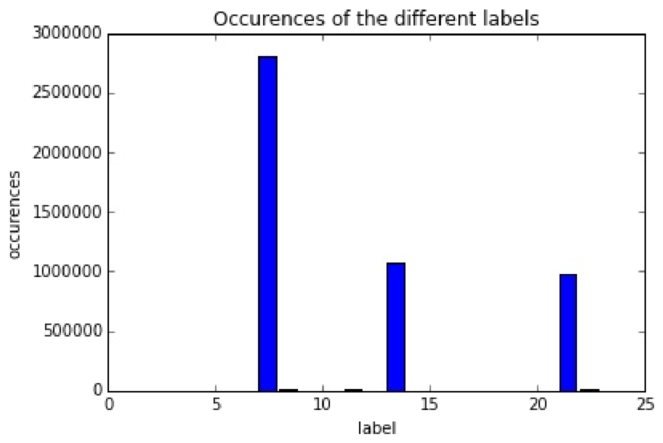
In [100]:

```
print('cluster label count')
clusterLabelCount = data.map(lambda x: (model.predict(x), 1)).reduceByKey(lambda a,b : a+b).collect()

for item in clusterLabelCount :
    print(item)


print('labelsAndData occurences')
wordsoccurences=labelsAndData.map(lambda x: (x[0], 1)).reduceByKey(lambda a,b: a+b).collect()
for item in wordsoccurences :
    print(item)
```

```
cluster label count
(0, 5)
(1, 4898426)
labelsAndData occurences
('guess_passwd.', 53)
('loadmodule.', 9)
('spy.', 2)
('phf.', 4)
('back.', 2203)
('pod.', 264)
('nmap.', 2316)
('smurf.', 2807886)
('satan.', 15892)
('warezmaster.', 20)
('land.', 21)
('ipsweep.', 12481)
('teardrop.', 979)
('neptune.', 1072017)
('multihop.', 7)
('perl.', 3)
('buffer_overflow.', 30)
('imap.', 12)
('warezclient.', 1020)
('rootkit.', 10)
('ftp_write.', 8)
('normal.', 972781)
('portsweep.', 10413)
```

In [101]:

```
Y=[wordsoccurences[i][1]for i in range (len(wordsoccurences))]
X=range(len(Y))
barlist =plt.bar(X,Y)
plt.title("Occurences of the different labels")
plt.xlabel("label")
plt.ylabel("occurences")
plt.show()
```



We runned this clustering algorithm several time, and the outcome is each time pretty different : In the last execution we got a bad clustering since one group contains only 5 elements and the other one contains more than 4 million elements. So we can conclude that, in one hand the clustering depend strongly on the initialization of the centroids, and on the other hand choosing to divide the data into two groups only is not accurate and not adapted to the nature of our data since it has 38 features

Let's see the repartition of labels in each cluster now .

In [49]:

```
labelprediction = labelsAndData.map(lambda x : (x[0],model.predict(x[1])))
firstcluster = labelprediction.filter(lambda x : x[1] == 0)
secondcluster = labelprediction.filter(lambda x : x[1] == 1)

print('labelsAndData occurences for the first cluster')
wordsoccurences0=firstcluster.map(lambda x: (x[0], 1)).reduceByKey(lambda a,b: a+b).collect()
for item in wordsoccurences0 :
    print(item)

print('labelsAndData occurences for the second cluster')
wordsoccurences1=secondcluster.map(lambda x: (x[0], 1)).reduceByKey(lambda a,b: a+b).collect()
for item in wordsoccurences1 :
    print(item)
```

```
labelsAndData occurences for the first cluster
('guess_passwd.', 53)
('loadmodule.', 9)
('phf.', 4)
('spy.', 2)
('nmap.', 2316)
('satan.', 15885)
('warezmaster.', 20)
('land.', 21)
('ipsweep.', 12481)
('teardrop.', 979)
('neptune.', 1072016)
('multihop.', 6)
('perl.', 3)
('buffer_overflow.', 30)
('imap.', 11)
('warezclient.', 897)
('rootkit.', 10)
('ftp_write.', 6)
('normal.', 857475)
('portsweep.', 10403)
labelsAndData occurences for the second cluster
('back.', 2203)
('smurf.', 2807886)
('pod.', 264)
('satan.', 7)
('multihop.', 1)
('neptune.', 1)
('imap.', 1)
('warezclient.', 123)
('ftp_write.', 2)
('normal.', 115306)
('portsweep.', 10)
```

2.2.4. Choosing K

How many clusters are appropriate for a dataset? In particular, for our own dataset, it's clear that there are 23 distinct behavior patterns in the data, so it seems that k could be at least 23, or likely, even more. In other cases, we even don't have any information about the number of patterns at all (remember, generally your data is not labelled!). Our task now is finding a good value of k. For doing that, we have to build and evaluate models with different values of k. A clustering could be considered good if each data point were near to its closest centroid. One of the ways to evaluate a model is calculating the Mean of Squared Errors of all data points.

Question 6

Complete the function below to calculate the MSE of each model that is corresponding to each value of k. Plot the results. From the obtained result, what is the best value for k ? Why ?

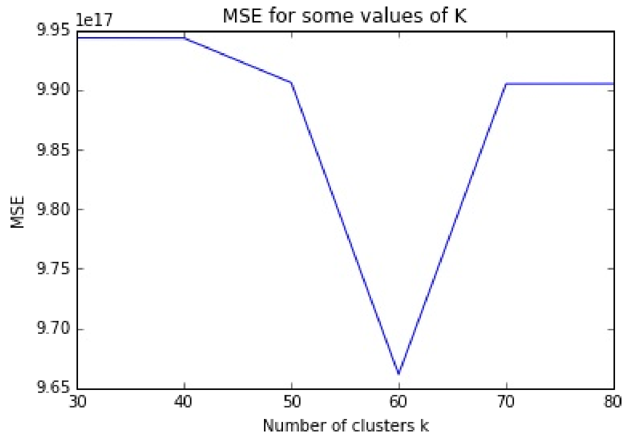
In [13]:

```python
# k: the number of clusters
def clusteringScore(data, k):
    print(k)
    clusters = KMeans.train(data, k, maxIterations=10,runs=10, initializationMode="random")
    # calculate mean square error
    return data.map(lambda x : error(clusters,x)).reduce(add)
```

In [25]:

```python
# k: the number of clusters
def clusteringScore(data, k):
    print(k)
    clusters = KMeans.train(data, k, maxIterations=10,runs=10, initializationMode="random")
    # calculate mean square error
    return data.map(lambda x : error(clusters,x)).reduce(add)

scores = [clusteringScore(data, k) for k in [30,40,50,60,70,80]]
for score in scores:
    print(score)

# plot results
listk=[30,40,50,60,70,80]
plt.plot(listk,scores)
plt.title("MSE for some values of K")
plt.xlabel("Number of clusters k")
plt.ylabel("MSE")
plt.show()
```

```
30

/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")

40
50
60
70
80
9.94337896398e+17
9.94326046661e+17
9.90600524836e+17
9.66135802618e+17
9.90489062861e+17
9.90499967316e+17
```



The plot above is logical, in fact when we increase the number of clusters we notice that the error has a decreasing pace eventhough it increases slightly between k=60 and k=70 which is probably due to repartition of points in each cluster which could lead to a higher distorsion

2.2.5 Normalizing features

K-means clustering treats equally all dimensions/directions of the space and therefore tends to produce more or less spherical (rather than elongated) clusters. In this situation, leaving variances uneven is equivalent to putting more weight on variables with smaller variance, so clusters will tend to be separated along variables with greater variance.

In our notebook, since Euclidean distance is used, the clusters will be influenced strongly by the magnitudes of the variables, especially by outliers. Normalizing will remove this bias.

Each feature can be normalized by converting it to a standard score. This means subtracting the mean of the feature's values from each value, and dividing by the standard deviation

$$normalize_i = \frac{feature_i - \mu_i}{\sigma_i}$$

Where,

- $normalize_i$ is the normalized value of feature i
- μ_i is the mean of feature i
- σ_i is the standard deviation of feature i

Question 7

Complete the code below to normalize the data. Print the first 5 lines of the new data.

HINT If $\sigma_i = 0$ then $normalize_i = feature_i - \mu_i$

In [15]:

```
from operator import add
def normalizeData(data):
    # number of connections
    n = data.count()

    # calculate the sum of each feature
    sums = data.reduce(add)
    print(sums)

    # calculate means
    means = sums/n

    # calculate the sum square of each feature

    sumSquares = data.map(lambda x: (x-means)**2 ).reduce(add)
    print(sumSquares)

    # calculate standard deviation of each feature
    stdevs = (sumSquares/n)**0.5
    print(stdevs)

    def normalize(point):
        ppoint=[]
        for k in range(len(stdevs)):
            if stdevs[k]==0:
                ppoint+=[point[k]-means[k]]
            else:
                ppoint+=[(point[k]-means[k])/stdevs[k]]
        return ppoint


    return data.map(normalize)

normalizedData = normalizeData(data).cache()
print(normalizedData.take(5))
```

```
[  2.36802060e+08   8.98676524e+09   5.35703589e+09   2.80000000e+01
   3.17800000e+03   3.90000000e+01   6.09250000e+04   1.57000000e+02
   7.03067000e+05   3.96200000e+04   3.34000000e+02   1.80000000e+02
   6.33610000e+04   5.82300000e+03   3.64000000e+02   5.00200000e+03
   0.00000000e+00   2.00000000e+00   4.09100000e+03   1.64084428e+09
   1.44634545e+09   8.71775140e+05   8.72101730e+05   2.82468470e+05
   2.82786920e+05   3.86919313e+06   1.03746840e+05   1.38433600e+05
   1.14124176e+09   9.26852923e+08   3.69201228e+06   1.50436230e+05
   2.96380553e+06   3.16639800e+04   8.72367200e+05   8.71361620e+05
   2.83755350e+05   2.82440660e+05]
[  2.56288804e+12   4.34144161e+18   2.03794728e+18   2.79998399e+01
   8.99593818e+03   2.54999689e+02   1.07736324e+06   2.60994968e+02
   6.02156483e+05   7.28516355e+07   3.33977226e+02   3.19993386e+02
   7.59669934e+07   7.55440779e+04   3.73972951e+02   6.17689224e+03
   0.00000000e+00   1.99999918e+00   4.08758334e+03   2.20135894e+11
   2.96415821e+11   7.14332989e+05   7.15749707e+05   2.64228272e+05
   2.65156244e+05   7.42363185e+05   3.35136000e+04   9.67782387e+04
   2.00770991e+10   5.49482085e+10   8.28196854e+05   5.77114862e+04
   1.13324779e+06   8.33893726e+03   7.14193132e+05   7.15462699e+05
   2.61255735e+05   2.61334681e+05]
[  7.23329737e+02   9.41430978e+05   6.45012268e+05   2.39083319e-03
   4.28543324e-02   7.21508295e-03   4.68978117e-01   7.29940683e-03
   3.50611523e-01   3.85648064e+00   8.25714534e-03   8.08243095e-03
   3.93807490e+00   1.24185736e-01   8.73758872e-03   3.55104777e-02
   0.00000000e+00   6.38978745e-04   2.88871577e-02   2.11990761e+02
   2.45992685e+02   3.81875553e-01   3.82254047e-01   2.32252900e-01
   2.32660379e-01   3.89295798e-01   8.27145751e-02   1.40559550e-01
   6.40209308e+01   1.05912757e+02   4.11185973e-01   1.08543203e-01
   4.80987669e-01   4.12597750e-02   3.81838168e-01   3.82177399e-01
   2.30942796e-01   2.30977686e-01]
[[-0.066833185424034519, -0.0017203822822947088, 0.068188435127257627, -0.0023908468608653793,
-0.015139173434069231, -0.0011034846181951323, -0.026520759954878015, -0.0043909155847286791, 2
.4427918743690413, -0.0020973278346766082, -0.0082577083974741533, -0.0045464613866545425, -0.0
032845891672013778, -0.0095723392157663588, -0.0085045784245815128, -0.028756112730921109, 0.0,
 -0.00063897900545473846, -0.028911303443075482, -1.575415074433681, -1.196243238095458, -0.466
04261387212803, -0.4657555739116494, -0.24828577490785206, -0.24813035170526898, 0.539733092700
60821, -0.25605652029026976, -0.20105929643768922, -3.6391392588237448, -1.7865104366018241, -1
.8330227339709937, -0.28293900018693519, -1.2579366383602955, -0.15666848795494887, -0.46640478
356620507, -0.46545364053023996, -0.25083182898658818, -0.24963196577649319], [-0.0668331854240
34519, -0.0017767955869251, 0.0053245145203899237, -0.0023908468608653793, -0.0151391734340692
31, -0.0011034846181951323, -0.026520759954878015, -0.0043909155847286791, 2.4427918743690413,
-0.0020973278346766082, -0.0082577083974741533, -0.0045464613866545425, -0.0032845891672013778,
 -0.0095723392157663588, -0.0085045784245815128, -0.028756112730921109, 0.0, -0.000638979005454
73846, -0.028911303443075482, -1.5706978877222004, -1.1921780765654681, -0.46604261387212803, -
0.4657555739116494, -0.24828577490785206, -0.24813035170526898, 0.53973309270060821, -0.2560565
2029026976, -0.20105929643768922, -3.6235193672155206, -1.777068703304864, 0.59896684321121529,
 -0.28293900018693519, 0.82111873926430345, -0.15666848795494887, -0.46640478356620507, -0.4654
5364053023996, -0.25083182898658818, -0.24963196577649319], [-0.066833185424034519, -0.00169807
58142880329, 0.0002083327604347245, -0.0023908468608653793, -0.015139173434069231, -0.001103484
6181951323, -0.026520759954878015, -0.0043909155847286791, 2.4427918743690413, -0.0020973278346
766082, -0.0082577083974741533, -0.0045464613866545425, -0.0032845891672013778, -0.009572339215
7663588, -0.0085045784245815128, -0.028756112730921109, 0.0, -0.00063897900545473846, -0.028911
303443075482, -1.575415074433681, -1.196243238095458, -0.46604261387212803, -0.4657555739116494
, -0.24828577490785206, -0.24813035170526898, 0.53973309270060821, -0.25605652029026976, -0.201
05929643768922, -3.6078994756072964, -1.7676269700079039, 0.59896684321121529, -0.2829390001869
3519, -0.21840894954799603, -0.15666848795494887, -0.46640478356620507, -0.46545364053023996, -
0.25083182898658818, -0.24963196577649319], [-0.066833185424034519, -0.0017012624525747008, 0.0
014548206801329004, -0.0023908468608653793, -0.015139173434069231, -0.0011034846181951323, -0.0
26520759954878015, -0.0043909155847286791, 2.4427918743690413, -0.0020973278346766082, -0.00825
77083974741533, -0.0045464613866545425, -0.0032845891672013778, -0.0095723392157663588, -0.0085
045784245815128, -0.028756112730921109, 0.0, -0.00063897900545473846, -0.028911303443075482, -1
.5706978877222004, -1.1921780765654681, -0.46604261387212803, -0.4657555739116494, -0.248285774
90785206, -0.24813035170526898, 0.53973309270060821, -0.25605652029026976, -0.20105929643768922
, -3.5922795839990722, -1.7581852367109438, 0.59896684321121529, -0.28293900018693519, -0.57184
83637441778, -0.15666848795494887, -0.46640478356620507, -0.46545364053023996, -0.2508318289865
8818, -0.24963196577649319], [-0.066833185424034519, -0.0016948891760013649, -0.000942032956500
65666, -0.0023908468608653793, -0.015139173434069231, -0.0011034846181951323, -0.02652075995487
8015, -0.0043909155847286791, 2.4427918743690413, -0.0020973278346766082, -0.008257708397474153
3, -0.0045464613866545425, -0.0032845891672013778, -0.0095723392157663588, -0.00850457842458151
28, -0.028756112730921109, 0.0, -0.00063897900545473846, -0.028911303443075482, -1.565980701010
7198, -1.1881129150354781, -0.46604261387212803, -0.4657555739116494, -0.24828577490785206, -0.
24813035170526898, 0.53973309270060821, -0.25605652029026976, -0.20105929643768922, -3.57665969
2390848, -1.7487435034139838, 0.59896684321121529, -0.28293900018693519, -0.7381727939541457, -
0.15666848795494887, -0.46640478356620507, -0.46545364053023996, -0.25083182898658818, -0.24963
196577649319]]
```

Question 8

Using the new data, build different models with different values of $k \in [60, 70, 80, 90, 100, 110]$. Evaluate the results by plotting them and choose the best value of k.

We have chosen to work here with the sqrt of the error in order not to get really high values of mean error , we could have taken the normal values or divide all the values by the total number of connections but this would lead to the same result = the same shape of the curve .

In [23]:

```
from math import sqrt
def error(clusters, point):
    closest_center = clusters.centers[clusters.predict(point)]
    return sqrt(np.power(euclidean_distance(closest_center, point), 2))
```

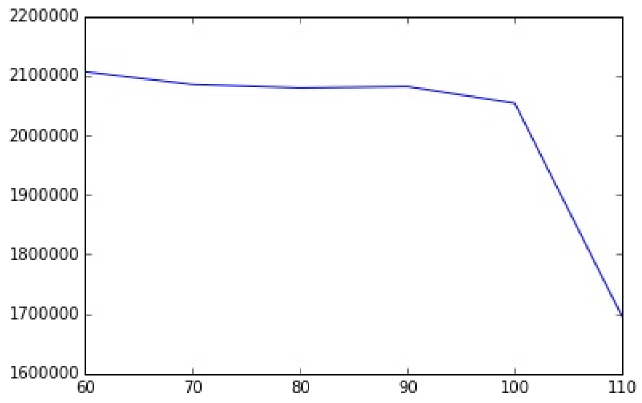
In [78]:

```
scores = [clusteringScore(normalizedData , k) for k in [60,70,80,90,100,110]]
for score in scores:
    print(score)

# plot results
listk=[60,70,80,90,100,110]
plt.plot(listk,scores)
```

```
60

/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")

70
80
90
100
110
2106440.5790530466
2085300.725645369
2079530.1668211909
2081690.7450067436
2053717.1368386722
1695618.1660101693
```

Out[78]:

```
[<matplotlib.lines.Line2D at 0x7f61a8880080>]
```



Note that here we used the function error by applying the function sqrt at the end :sqrt(np.power(euclidean_distance(closest_center, point), 2)) so that we will not get really high values , we can notice that when k gets higher the MSE falls down which is logical since the distorsion goes down , we found a good value in term of minimizing the error which is 110 that we will use fore later models .

Question 9

Plot the clustering result to see the difference between before and after normalizing features. Discuss about the difference and explain why and if normalization was useful.

In [78]:

```
print('cluster label count')
clusters = KMeans.train(normalizedData, 2, maxIterations=10,runs=10, initializationMode="random")
clusterLabelCountt = normalizedData.map(lambda x: (clusters.predict(x), 1)).reduceByKey(lambda a,b : a+b).co
llect()

for item in clusterLabelCountt :
    print(item)
```

```
cluster label count

/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")

(0, 4075896)
(1, 822535)
```

In [33]:

```
print('cluster label count')
clusters = KMeans.train(normalizedData, 110, maxIterations=10,runs=10, initializationMode="random")
clusterLabelCountt = normalizedData.map(lambda x: (clusters.predict(x), 1)).reduceByKey(lambda a,b : a+b).co
llect()

for item in clusterLabelCountt :
    print(item)
```

```
cluster label count

/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")
```

```
(0, 14685)
(24, 996)
(12, 28562)
(48, 80742)
(36, 62576)
(1, 12275)
(37, 41464)
(13, 39541)
(49, 52036)
(25, 343927)
(26, 970)
(2, 117185)
(50, 70499)
(38, 44272)
(14, 4328)
(27, 32594)
(51, 5495)
(3, 689)
(39, 15750)
(15, 4095)
(16, 4852)
(28, 30591)
(4, 25874)
(52, 13755)
(40, 75256)
(17, 34757)
(41, 2386)
(53, 39663)
(5, 123859)
(29, 4016)
(54, 31857)
(18, 39559)
(42, 103478)
(30, 56132)
(6, 52178)
(31, 207150)
(55, 9912)
(19, 12168)
(43, 21351)
(7, 2267984)
(8, 41990)
(32, 29546)
(20, 61813)
(56, 54714)
(44, 30968)
(33, 22212)
(57, 33913)
(9, 27033)
(21, 16252)
(45, 208492)
(10, 25252)
(46, 19819)
(34, 66944)
(22, 25669)
(23, 12992)
(35, 44793)
(11, 41985)
(47, 4585)
```

In [35]:

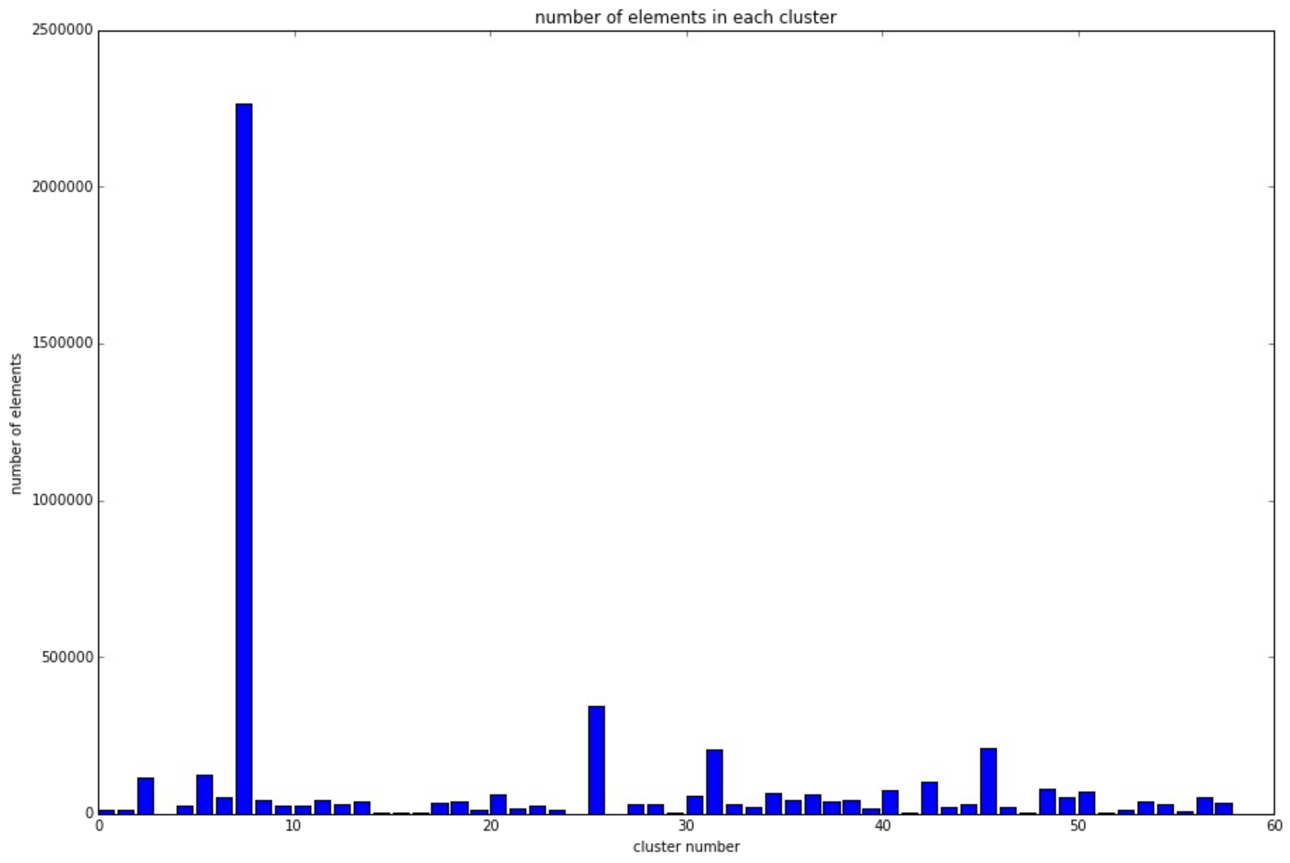
```
wordsoccurences[0][0]
```

Out[35]:

```
'guess_passwd.'
```

In [40]:

```
X=[clusterLabelCountt[i][0]for i in range (len(clusterLabelCountt))]
Y=[clusterLabelCountt[i][1]for i in range (len(clusterLabelCountt))]
plt.figure(figsize=(15,10))
barlist =plt.bar(X,Y)
plt.title("number of elements in each cluster")
plt.xlabel("cluster number")
plt.ylabel("number of elements")
plt.show()
```



Let us compute the repartition of our data in the previous model using data instead of normalised data in order to make a comparison

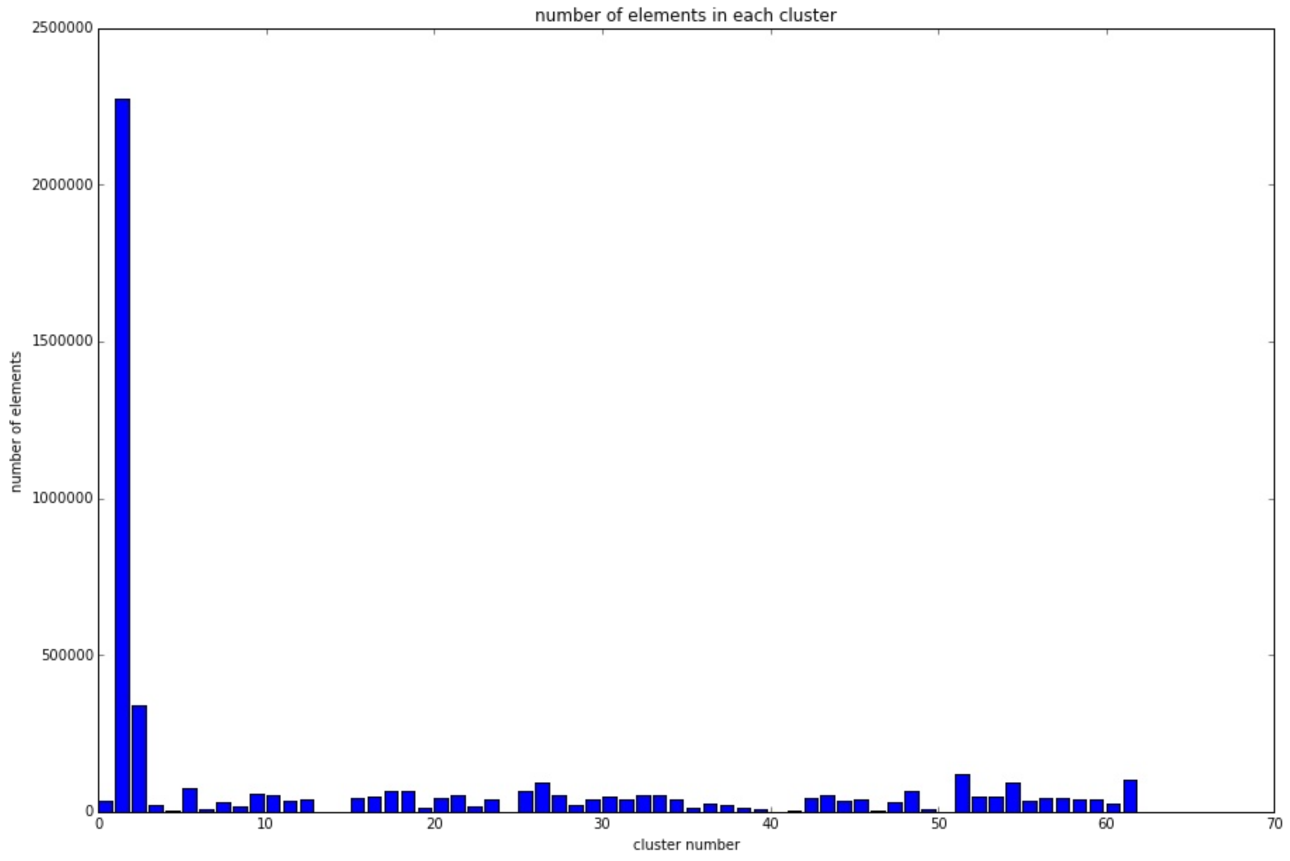
In [104]:

```
print('cluster label count')
clusters = KMeans.train(data, 110, maxIterations=10,runs=10, initializationMode="random")
clusterLabelCountt = data.map(lambda x: (clusters.predict(x), 1)).reduceByKey(lambda a,b : a+b).collect()
```

```
cluster label count

/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")
```

In [105]:

```
X=[clusterLabelCountt[i][0]for i in range (len(clusterLabelCountt))]
Y=[clusterLabelCountt[i][1]for i in range (len(clusterLabelCountt))]
plt.figure(figsize=(15,10))
barlist =plt.bar(X,Y)
plt.title("number of elements in each cluster")
plt.xlabel("cluster number")
plt.ylabel("number of elements")
plt.show()
```



We notice first that there is one cluster in both our models (using data and using normalised data) which is containing a huge number of elements , however we can see that for data the clusters except the biggest one contain more or less the same amount of elements whereas for normalised data some clusters contain way more data than others , which is a good clustering when for example in google search we have more information for a certain query than for others we can feel like there has been a better clustering .

In [30]:

```
def error(clusters, point):
    closest_center = clusters.centers[clusters.predict(point)]
    return np.sum([x**2 for x in (point - closest_center)])
```

In [31]:

```
WSSSEdata = data.map(lambda x : error(clusters,x)).reduce(add)
print("Within Set Sum of Squared Error for data = " + str(WSSSEdata))

WSSSEnormalizeddata = normalizedData.map(lambda x : error(clusters,x)).reduce(add)
print("Within Set Sum of Squared Error for normalized data  = " + str(WSSSEnormalizeddata))
```

```
Within Set Sum of Squared Error for data = 6.37941578928e+18
Within Set Sum of Squared Error for normalized data  = 56650897.5022
```

An other approach would be to calculate the errors for both data and normalizeddata , we can see that for normalized data it is much smaller with a value of 56650897.5022 comparing to 6.37941578928e+18 for the data that is not normalized

2.3. Clustering using categorical features

2.3.1 Loading data

In the previous section, we ignored the categorical features of our data: this is not a good idea, since these categorical features can be important in providing useful information for clustering. The problem is that K-means (or at least, the one we have developed and the one we use from MLLib) only work with data points in a metric space. Informally, this means that operations such as addition, substraction and computing the mean of data points are trivial and well defined. For a more formal definition of what a metric space is, follow this link (https://en.wikipedia.org/wiki/Metric_space#Definition).

What we will do next is to transform each categorical features into one or more numerical features. This approach is very widespread: imagine for example you wanted to use K-means to cluster text data. Then, the idea is to transform text data in d-dimensional vectors, and a nice way to do it is to use word2vec (http://deeplearning4j.org/word2vec). If you're interested, follow this link to a nice blog post (http://bigdatasciencebootcamp.com/posts/Part_3/clustering_news.html) on the problem.

There are two approaches:

- **Approach 1**: mapping **one** categorial feature to **one** numerical feature. The values in each categorical feature are encoded into unique numbers of the new numerical feature. For example, ['VERY HOT','HOT', 'COOL', 'COLD', 'VERY COLD'] will be encoded into [0,1,2,3,4,5]. However, by using this method, we implicit assume that the value of 'VERY HOT' is smaller than 'HOT'... This is not generally true.
- **Approach 2**: mapping mapping **one** categorial feature to **multiple** numerical features. Basically, a single variable with n observations and d distinct values, to d binary variables with n observations each. Each observation indicating the presence (1) or absence (0) of the d^{th} binary variable. For example, ['house', 'car', 'tooth', 'car'] becomes

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

We call the second approach "one-hot encoding". By using this approach, we keep the same role for all values of categorical features.

Question 10

Calculate the number of distinct categorical features value (at index `1,2,3`). Then construct a new input data using one-hot encoding for these categorical features (don't throw away numerical features!).

In [16]:

```
# c: index of the column
def getValuesOfColumn(data, c):
    return data.map(lambda x: x[c]).distinct().collect()

vColumn1 = getValuesOfColumn(raw_data, 1)
numValuesColumn1 = len(vColumn1)
vColumn1 = dict(zip(vColumn1, range(0, numValuesColumn1)))


vColumn2 = getValuesOfColumn(raw_data, 2)
numValuesColumn2 = len(vColumn2)
vColumn2 = dict(zip(vColumn2, range(0, numValuesColumn2)))



vColumn3 = getValuesOfColumn(raw_data, 3)
numValuesColumn3 = len(vColumn3)
vColumn3 = dict(zip(vColumn3, range(0, numValuesColumn3)))


def parseLineWithHotEncoding(line):
    cols = line.split(',')
    # label is the last column
    label = cols[-1]

    vector = cols[0:-1]
    featureOfCol1 = [0]*numValuesColumn1
    featureOfCol2 = [0]*numValuesColumn2
    featureOfCol3 = [0]*numValuesColumn3
    featureOfCol1[vColumn1[vector[1]]] = 1
    featureOfCol2[vColumn2[vector[2]]] = 1
    featureOfCol3[vColumn3[vector[3]]] = 1

    vector = ([vector[0]] + featureOfCol1 + featureOfCol2 +
        featureOfCol3 + vector[4:])

    # convert each value from string to float
    vector = np.array(list(map(lambda x: float(x), vector)))

    return (label, vector)

labelsAndData = raw_data.map(parseLine)

# we only need the data, not the label
data = labelsAndData.values().cache()
```

In [17]:

```
normalizedData = normalizeData(data).cache()
```

```
[  2.36802060e+08   8.98676524e+09   5.35703589e+09   2.80000000e+01
   3.17800000e+03   3.90000000e+01   6.09250000e+04   1.57000000e+02
   7.03067000e+05   3.96200000e+04   3.34000000e+02   1.80000000e+02
   6.33610000e+04   5.82300000e+03   3.64000000e+02   5.00200000e+03
   0.00000000e+00   2.00000000e+00   4.09100000e+03   1.64084428e+09
   1.44634545e+09   8.71775140e+05   8.72101730e+05   2.82468470e+05
   2.82786920e+05   3.86919313e+06   1.03746840e+05   1.38433600e+05
   1.14124176e+09   9.26852923e+08   3.69201228e+06   1.50436230e+05
   2.96380553e+06   3.16639800e+04   8.72367200e+05   8.71361620e+05
   2.83755350e+05   2.82440660e+05]
[  2.56288804e+12   4.34144161e+18   2.03794728e+18   2.79998399e+01
   8.99593818e+03   2.54999689e+02   1.07736324e+06   2.60994968e+02
   6.02156483e+05   7.28516355e+07   3.33977226e+02   3.19993386e+02
   7.59669934e+07   7.55440779e+04   3.73972951e+02   6.17689224e+03
   0.00000000e+00   1.99999918e+00   4.08758334e+03   2.20135894e+11
   2.96415821e+11   7.14332989e+05   7.15749707e+05   2.64228272e+05
   2.65156244e+05   7.42363185e+05   3.35136000e+04   9.67782387e+04
   2.00770991e+10   5.49482085e+10   8.28196854e+05   5.77114862e+04
   1.13324779e+06   8.33893726e+03   7.14193132e+05   7.15462699e+05
   2.61255735e+05   2.61334681e+05]
[  7.23329737e+02   9.41430978e+05   6.45012268e+05   2.39083319e-03
   4.28543324e-02   7.21508295e-03   4.68978117e-01   7.29940683e-03
   3.50611523e-01   3.85648064e+00   8.25714534e-03   8.08243095e-03
   3.93807490e+00   1.24185736e-01   8.73758872e-03   3.55104777e-02
   0.00000000e+00   6.38978745e-04   2.88871577e-02   2.11990761e+02
   2.45992685e+02   3.81875553e-01   3.82254047e-01   2.32252900e-01
   2.32660379e-01   3.89295798e-01   8.27145751e-02   1.40559550e-01
   6.40209308e+01   1.05912757e+02   4.11185973e-01   1.08543203e-01
   4.80987669e-01   4.12597750e-02   3.81838168e-01   3.82177399e-01
   2.30942796e-01   2.30977686e-01]
```

2.3.2. Building models

Question 11

Using the new data, cluster the connections with different values of $k \in [80, 90, 100, 110, 120, 130, 140, 150, 160]$. Evaluate the results and choose the best value of k as previous questions.

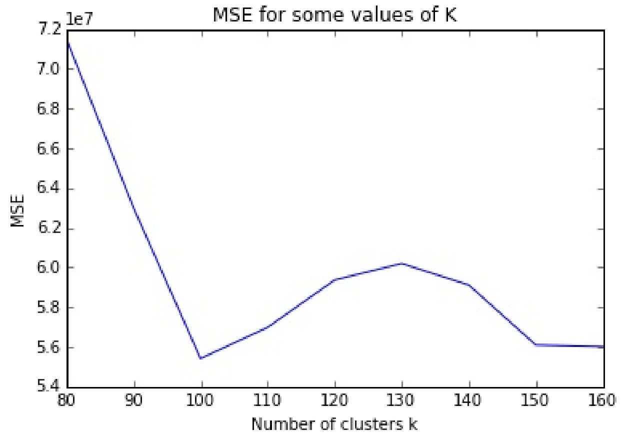
In [24]:

```
scores = [clusteringScore(normalizedData , k) for k in [80,90,100,110,120,130,140,150,160]]
for score in scores:
    print(score)

# plot results
listk=[80,90,100,110,120,130,140,150,160]
plt.plot(listk,scores)
plt.title("MSE for some values of K")
plt.xlabel("Number of clusters k")
plt.ylabel("MSE")
plt.show()
```

```
80
```

```
/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")
```

```
90
100
110
120
130
140
150
160
71500156.4621
63009581.3182
55408907.0033
56977967.1439
59365009.5328
60189752.3484
59116734.3167
56092925.4484
56007781.3797
```



We can notice that the error has generally a decreasing pace, eventhough is increases slightly between k = 100 and k = 150 a good value of k could be then from 100 to 150 .

2.4. Anomaly detection

When we have a new connection data (e.g., one that we never saw before), we simply find the closest cluster for it, and use this information as a proxy to indicate whether the data point is anomalous or not. A simple approach to decide wheter there is an anomaly or not, amounts to measuring the new data point’s distance to its nearest centroid. If this distance exceeds some thresholds, it is anomalous.

Question 12

Build your model with the best value of k in your opinion. Then, detect the anomalous connections in our data. Plot and discuss your result.

HINT The threshold has strong impact on the result. Be careful when choosing it! A simple way to choose the threshold's value is picking up a distance of a data point from among known data. For example, the 100th-farthest data point distance can be an option.

In [18]:

```
clusters = KMeans.train(normalizedData,110, maxIterations=10,runs=10, initializationMode="random")
```

```
/opt/spark/python/pyspark/mllib/clustering.py:347: UserWarning: The param `runs` has no effect
since Spark 2.0.0.
  warnings.warn("The param `runs` has no effect since Spark 2.0.0.")
```

In [19]:

```
centroid=clusters.clusterCenters
```

In [40]:

```
len(centroid)
```

Out[40]:

```
54
```

In [41]:

```
distances = normalizedData.map(lambda x: euclidean_distance(x, centroid[clusters.predict(x)])).sortBy(lambda
 x : -x).collect()
```

In [52]:

```
Threshold=distances[100]
print(Threshold)
```

```
297.097381809
```

In [53]:

```
def anomalousclassifier(clusters,centroid,point,Threshold):
    if euclidean_distance(point, centroid[clusters.predict(point)])>Threshold :
        print('this point is anomalous')
    else :
        print('this point is not anomalous')
```

In [56]:

```
point=normalizedData.take(10000)[1000]
anomalousclassifier(clusters,centroid,point,Threshold)
```

```
this point is not anomalous
```

In [67]:

```
import random
randompoint=random.sample(range(100), 38)
anomalousclassifier(clusters,centroid,randompoint,Threshold)
```

```
this point is anomalous
```

We tried to take a random point and tested if it is anomalous , we found that it is anoumalous which is logical since we are randomizing 38 features , therefore the normal thing is that our point be far from all the clusters . Then we took a point from our data and tested our function into it , we found that it is not anomalous which normal since only 100 points in the whole data are supposed to be anomalous (by fixing the threeshold)

Question 13

Try other methods to find the best value for k such as `silhouette`, `entropy`... In particular, with this data, you can take advantage of predefined labels to calculate the quality of model using entropy... However, we suggest you to try with `silhouette`. It's more general and can work with any dataset (with and without predefined labels).

Here are some additional information about the metrics we suggest to use:

- Silhouette (https://en.wikipedia.org/wiki/Silhouette_(clustering))
- Hack approach to Silhouette (http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html)
- Entropy (http://scikit-learn.org/stable/modules/clustering.html) [Lookup for entropy]

Note you are free to play with any relevant evaluation metric you think appropriate for your work!

In this part , we will try to write a silhouette function that is supposes to fit for the whole data , we will test it for one single point , however in order to use our silhouette function to our big data , we can not user data.map(function) when function already contains a map to calculate one of its variables , that is why we have chosen to adapt this function by approximating one of the variables (a =the mean distance between a point and all the point in the same cluster with it) by only the distance between this point and the center of its cluster . Note that this approximation is still basic since for example if many points have more or less the same position , our variable a will be null while our approximation is not .

In [109]:

```
centroid=clusters.clusterCenters
```

In [112]:

```
def silhouett(point):
    l=[]
    prediction=clusters.predict(point)
    distannce=normalizedData.filter(lambda x : clusters.predict(x)==prediction).map(lambda x :euclidean_dist
ance(x, point)).mean()
    listdistance=[euclidean_distance(x, point) for x in centroid]
    listdistance.sort()
    b=listdistance[1]
    a=distannce
    l+=[(b-a)/np.max([a,b])]
    return np.mean(l)
```

In [107]:

```
point=normalizedData.take(10000)[1000]
```

In [113]:

```
silhouett(point)
```

Out[113]:

```
0.96345623557174609
```

Our approximation for silhouette

In [20]:

```
def adaptedsilhouett(point):
    l=[]
    prediction=clusters.predict(point)
    distannce=euclidean_distance(point, centroid[clusters.predict(point)])
    listdistance=[euclidean_distance(x, point) for x in centroid]
    listdistance.sort()
    b=listdistance[1]
    a=distannce
    l+=[(b-a)/np.max([a,b])]
    return np.mean(l)
```

In [21]:

```
silhouetteapproximationvalue=normalizedData.map(adaptedsilhouett).mean()
```

In [22]:

```
print(silhouetteapproximationvalue)
```

```
0.687936908995
```

Question 14

Implement K-means on Spark so that It can work with large datasets in parallel. Test your algorithm with our dataset in this notebook. Compare our algorithm with the algorithm from MLLIB.
Let's clarify the meaning of this question: what we want is for students to design the K-means algorithm for the parallel programming model exposed by Spark. You are strongly invited to use the Python API (pyspark). So, at the end of the day, you will operate on RDDs, and implement a `map/reduce` algorithm that performs the two phases of the standard K-means algorithm, i.e. the assignment step and the update step.

In [ ]:

```
np.random.seed(22324)
import random
def kmeans_big_data(data, k, centroids=None):

    # randomize initial centroids

    centroids = random.sample(list(data),k)
    while True:

        iterations += 1

        old_centroids = centroids

        closeCentroid = data.map(lambda p: (find_closest_centroid(p, centroids), (p, 1)))

        pointStats = closeCentroid.reduceByKey(lambda (x1, y1), (x2, y2): (x1 + x2, y1 + y2))

        centroids = pointStats.map(lambda (x, (y, z)): (x, y / z)).collect()

        # if the stop criteria are met, stop the algorithm

        if check_converge(centroids, old_centroids, iterations, threshold=0.001):

            break

    return centroids
```

We tried to implement a K-means algorithm that can work on huge datasets. For this, we programmed the assignment and the update step using a map reduce algorithm. For luck of time, we didn't run it.