

Clustering techniques detecting atypical network connections

We refer to a dataset downloaded from the 1999 KDD cup competition. We start by sampling from all available data so as to be able to work interactively.

In [343]:

```
raw_data = sc.textFile('data/kddcup.data_10_percent.gz').sample(False, .01)
```

How many data do we have?

In [344]:

```
raw_data.count()
```

Out[344]:

4854

What kind of data do we have?

In [345]:

```
raw_data.first()
```

Out[345]:

```
u'0,tcp,http,SF,370,520,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,3,3,0.0  
0,0.00,0.00,0.00,1.00,0.00,0.00,72,255,1.00,0.00,0.01,0.04,0.0  
0,0.00,0.00,0.00,normal.'
```

Each element is a CSV (comma separated values) gathering 38 elements, mostly numeric ones. Let's import this data in a python list and peek through the first ten items.

In [346]:

```
original_data = raw_data.collect()
original_data[:10]
```

Out[346]:

```
[u'0,tcp,http,SF,370,520,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,3,3,0.0
0,0.00,0.00,0.00,1.00,0.00,0.00,72,255,1.00,0.00,0.01,0.04,0.0
0,0.00,0.00,0.00,normal.',
 u'0,tcp,http,SF,307,1377,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,8,8,0.0
0,0.00,0.00,0.00,1.00,0.00,0.00,8,255,1.00,0.00,0.12,0.06,0.0
0,0.00,0.00,0.00,normal.',
 u'0,tcp,http,SF,226,7430
1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,4,4,0.00,0.00,0.00,0.00,1.0
0,0.00,0.00,15,255,1.00,0.00,0.07,0.04,0.00,0.00,0.00,0.00,norma
l.',
 u'0,tcp,http,SF,224,2531,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,2,3,0.0
0,0.00,0.00,0.00,1.00,0.00,0.67,166,166,1.00,0.00,0.01,0.00,0.0
0,0.00,0.00,0.00,normal.',
 u'0,tcp,http,SF,307,891,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,8,11,0.0
0,0.00,0.00,0.00,1.00,0.00,0.18,255,255,1.00,0.00,0.00,0.00,0.0
0,0.00,0.00,0.00,normal.',
 u'0,tcp,http,SF,203,6312,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,5,5,0.0
0,0.00,0.00,0.00,1.00,0.00,0.00,21,255,1.00,0.00,0.05,0.02,0.0
0,0.00,0.00,0.00,normal.',
 u'0,tcp,http,SF,301,544,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,5,5,0.0
0,0.00,0.00,0.00,1.00,0.00,0.00,14,255,1.00,0.00,0.07,0.04,0.0
0,0.00,0.00,0.00,normal.',
 u'0,tcp,http,SF,196,170,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,4,5,0.0
0,0.00,0.00,0.00,1.00,0.00,0.40,85,255,1.00,0.00,0.01,0.01,0.0
0,0.00,0.00,0.00,normal.',
 u'0,tcp,http,SF,292,2808,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0.0
0,0.00,0.00,0.00,1.00,0.00,0.00,68,255,1.00,0.00,0.01,0.01,0.0
0,0.00,0.00,0.00,normal.',
 u'0,tcp,http,SF,309,6090,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,2,2,0.0
0,0.00,0.00,0.00,1.00,0.00,0.00,6,255,1.00,0.00,0.17,0.01,0.0
0,0.00,0.00,0.00,normal.']]
```

Each data item is represented by a string, but we want to be able to access to the single values deparated by commas, thus we start by splitting at each comma.

```
original_data[0].split(',')
```

```
[u'0',
u'tcp',
u'http',
u'SF',
u'370',
u'520',
u'0',
u'0',
u'0',
u'0',
u'0',
u'1',
u'0',
u'0',
u'0',
u'0',
u'0',
u'0',
u'0',
u'0',
u'0',
u'0',
u'0',
u'0',
u'3',
u'3',
u'0.00',
u'0.00',
u'0.00',
u'0.00',
u'1.00',
u'0.00',
u'0.00',
u'72',
u'255',
u'1.00',
u'0.00',
u'0.01',
u'0.04',
u'0.00',
u'0.00',
u'0.00',
u'0.00',
u'normal.']
```

<http://localhost:8888/notebooks/SPES/clustering/network-attack-clustering.ipynb>

In [348]:

```
def clean(line):
    splitted_line = line.split(',')
    return [float(splitted_line[0])] + splitted_line[1:4] + [float(d) for d in
clean(original_data[0])
```

Out[348]:

```
[0.0,
 u'tcp',
 u'http',
 u'SF',
 520.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 1.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 3.0,
 3.0,
 0.0,
 0.0,
 0.0,
 0.0,
 1.0,
 0.0,
 0.0,
 72.0,
 255.0,
 1.0,
 0.0,
 0.01,
 0.04,
 0.0,
 0.0,
 0.0,
 0.0,
 u'normal.' ]
```

We can now map the cleaning function to each line in the original dataset, in order to obtain a cleaned data set to be further processed.

In [349]:

```
data = map(clean, original_data)
```

Let's explore how some of the quantities in the dataset behave. For instance, if for each element we consider the 30-th and the 31-th elements and interpret them as coordinates of a point, we can describe this element through a point in the plane:

In [77]:

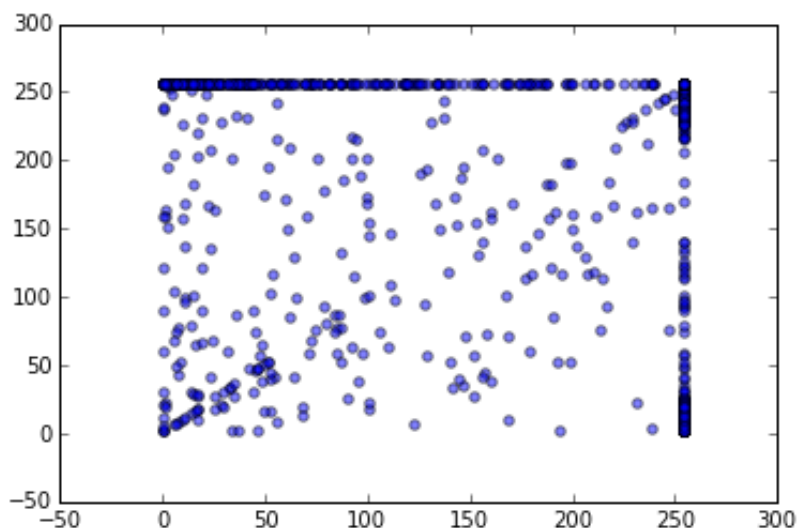
```
%matplotlib inline
import matplotlib.pyplot as plt

x_index, y_index = (30, 31)
x_points = [d[x_index] for d in data]
y_points = [d[y_index] for d in data]

plt.scatter(x_points, y_points, alpha=0.5)
```

Out[77]:

<matplotlib.collections.PathCollection at 0x10dbe62d0>



We can see some interesting structure. Let's organize in order to be able to quickly check what happens if we focus on different components of each element in the dataset.

In [78]:

```
def scatter_plot(data, x_index, y_index):

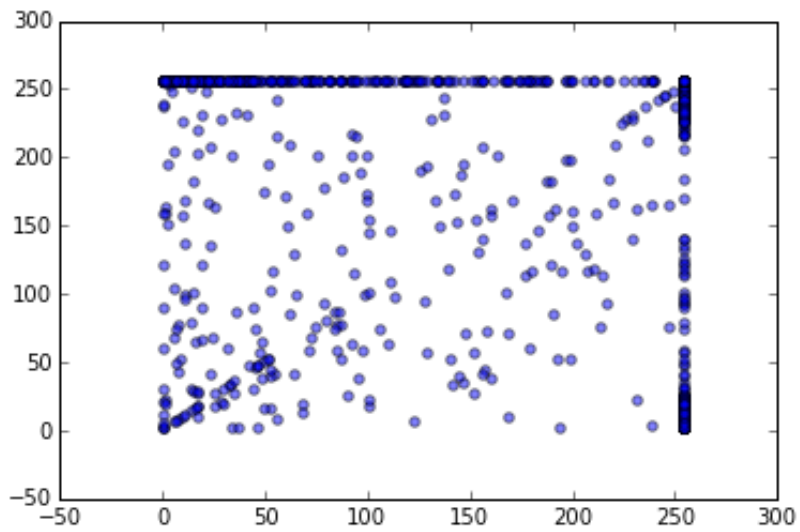
    x_points = [d[x_index] for d in data]
    y_points = [d[y_index] for d in data]
    return plt.scatter(x_points, y_points, alpha=0.5)
```

In [68]:

```
scatter_plot(data, 30, 31)
```

Out[68]:

<matplotlib.collections.PathCollection at 0x10e0d2810>

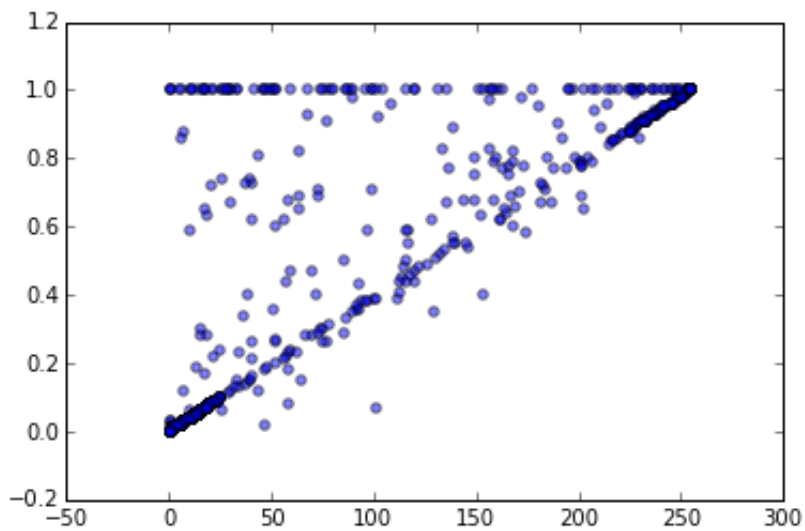


In [69]:

```
scatter_plot(data, 31, 32)
```

Out[69]:

<matplotlib.collections.PathCollection at 0x10e229410>

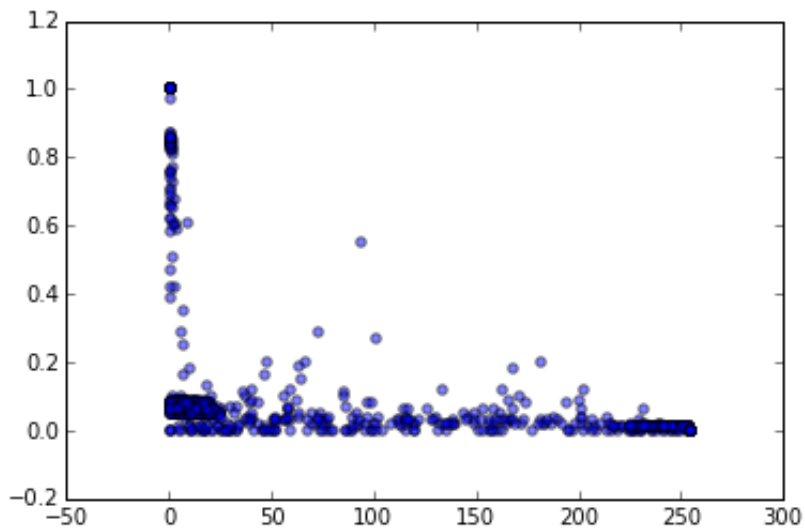


In [70]:

```
scatter_plot(data, 31, 33)
```

Out[70]:

```
<matplotlib.collections.PathCollection at 0x10e37b5d0>
```

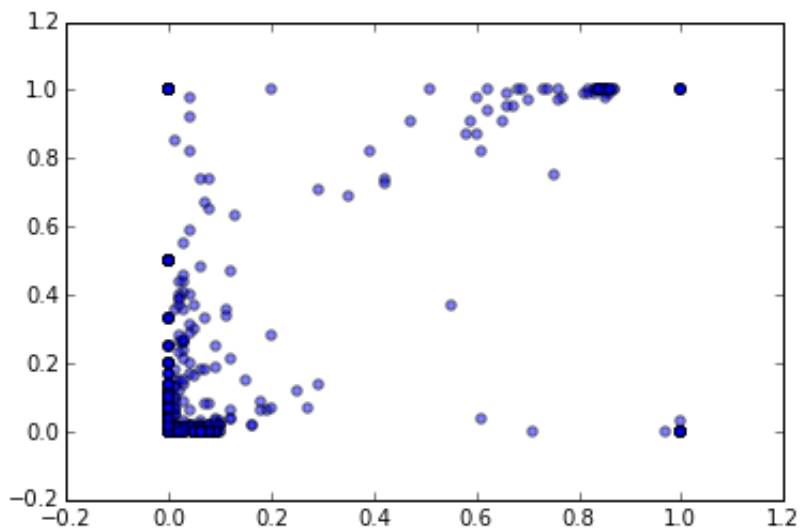


In [71]:

```
scatter_plot(data, 33, 34)
```

Out[71]:

```
<matplotlib.collections.PathCollection at 0x10e4cf750>
```



We see that points in the last graph tend to concentrate in different regions. We can apply a clustering algorithm in order to get more information about such regions.

The mathematical condition for the K clusters C_k and the K centroids μ_k can be expressed as:

$$\text{Minimize } \sum_{k=1}^K \sum_{x_n \in C_k} \|x_n - \mu_k\|^2 \text{ with respect to } C_k, \mu_k$$

An iterative method known as Lloyd's algorithm exists that converges (albeit to a local minimum) in few steps. The procedure alternates between two operations.

- Once a set of centroids μ_k is available, the clusters are updated to contain the points closest in distance to each centroid:

$$C_k = \{x_n : \|x_n - \mu_k\| \leq \text{all } \|x_n - \mu_l\|\}$$

- Given a set of clusters, the centroids are recalculated as the means of all points belonging to a cluster:

$$\mu_k = \frac{1}{C_k} \sum_{x_n \in C_k} x_n$$

The two-step procedure continues until the assignments of clusters and centroids no longer change.

In [356]:

```
import numpy as np
import random

from datetime import datetime
random.seed(datetime.now())

def cluster_points(X, mu):
    clusters = {}
    for x in X:
        bestmukey = min([(i[0], np.linalg.norm(x-mu[i[0]])) \
                        for i in enumerate(mu)], key=lambda t:t[1])[0]
        try:
            clusters[bestmukey].append(x)
        except KeyError:
            clusters[bestmukey] = [x]
    return clusters
```

In [357]:

```
def reevaluate_centers(mu, clusters):
    newmu = []
    keys = sorted(clusters.keys())
    for k in keys:
        newmu.append(np.mean(clusters[k], axis = 0))
    return newmu
```

In [358]:

```
def has_converged(mu, oldmu):
    return (set([tuple(a) for a in mu]) == set([tuple(a) for a in oldmu]))
```

In [359]:

```

def find_centers(X, K):
    # Initialize to K random centers
    oldmu = random.sample(X, K)
    mu = random.sample(X, K)
    while not has_converged(mu, oldmu):
        oldmu = mu
        # Assign all points in X to clusters
        clusters = cluster_points(X, mu)
        # Reevaluate centers
        mu = reevaluate_centers(oldmu, clusters)

    return(mu, clusters)

```

In [363]:

```

K = 2
X = [np.array( (d[33], d[34]) ) for d in data]

oldmu = random.sample(X, K)
mu = random.sample(X, K)
clusters = cluster_points(X, mu)
mu = reevaluate_centers(oldmu, clusters)

```

In [368]:

clusters[1]

Out[368]:

```

[array([ 0. ,  0.01]),
 array([ 0. ,  0.12]),
 array([ 0. ,  0.07]),
 array([ 0. ,  0.01]),
 array([ 0.,  0.]),
 array([ 0. ,  0.05]),
 array([ 0. ,  0.07]),
 array([ 0. ,  0.01]),
 array([ 0. ,  0.01]),
 array([ 0. ,  0.17]),
 array([ 0. ,  0.04]),
 array([ 0. ,  0.01]),
 array([ 0.,  0.]),
 array([ 0. ,  0.08]),
 array([ 0.,  0.]),
 array([ 0. ,  0.1]),
 array([ 0. ,  0.01]),
 array([ 0. ,  0.04]),

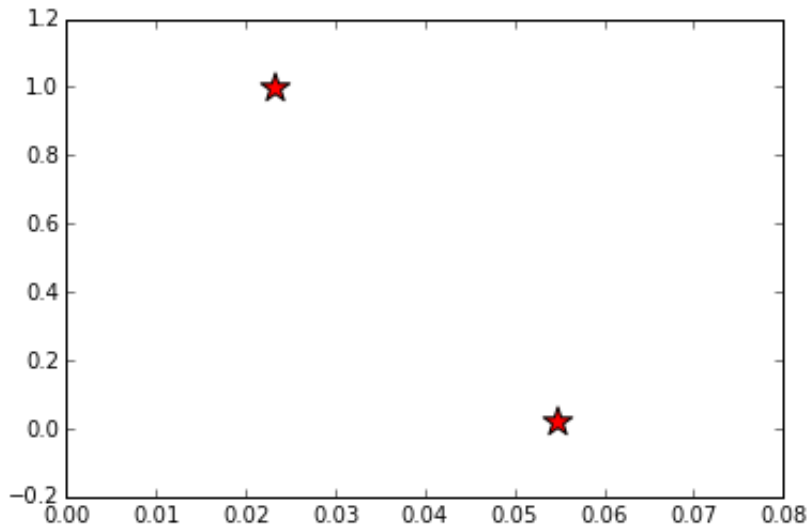
```

In [365]:

```
mu_x = [m[0] for m in mu]
mu_y = [m[1] for m in mu]
plt.scatter(mu_x, mu_y, marker="*", s=200, c='red')
```

Out[365]:

<matplotlib.collections.PathCollection at 0x112bff2d0>



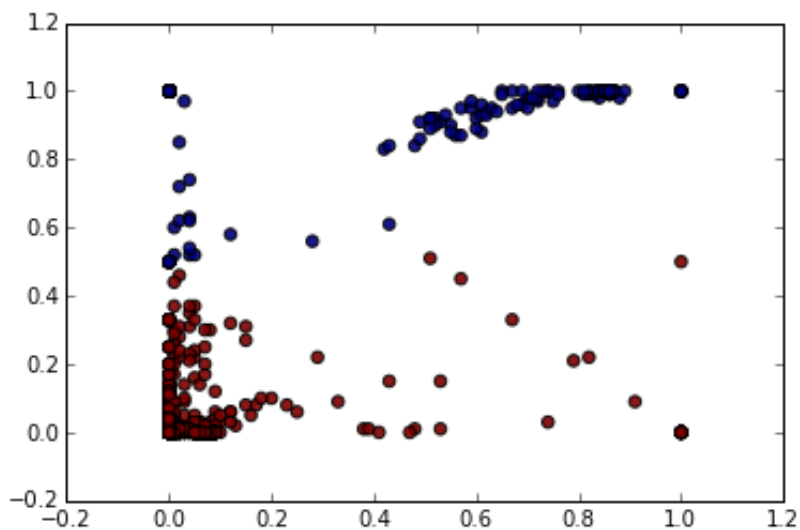
In [369]:

```
points_x = [d[0] for k in clusters.keys() for d in clusters[k] ]
points_y = [d[1] for k in clusters.keys() for d in clusters[k] ]

colors = [1.*k/len(clusters.keys()) for k in clusters.keys() for d in clusters[k] ]
plt.scatter(points_x, points_y, alpha=0.9, c=colors, s=30)
```

Out[369]:

<matplotlib.collections.PathCollection at 0x112c80610>



In [370]:

```
def show_clustering(mu, clusters):
    mu_x = [m[0] for m in mu]
    mu_y = [m[1] for m in mu]

    points_x = [d[0] for k in clusters.keys() for d in clusters[k] ]
    points_y = [d[1] for k in clusters.keys() for d in clusters[k] ]

    colors = [1.*k/len(clusters.keys()) for k in clusters.keys() for d in clusters[k] ]

    plt.scatter(points_x, points_y, alpha=0.9, c=colors, s=30)
    plt.scatter(mu_x, mu_y, marker="*", s=200, c='red')

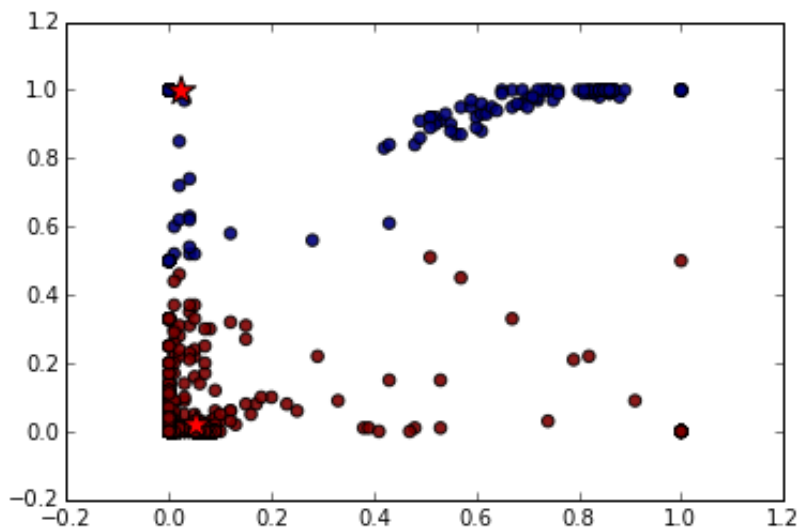
    return plt
```

In [371]:

```
show_clustering(mu, clusters)
```

Out[371]:

```
<module 'matplotlib.pyplot' from '/usr/local/lib/python2.7/site-packages/matplotlib/pyplot.pyc'>
```

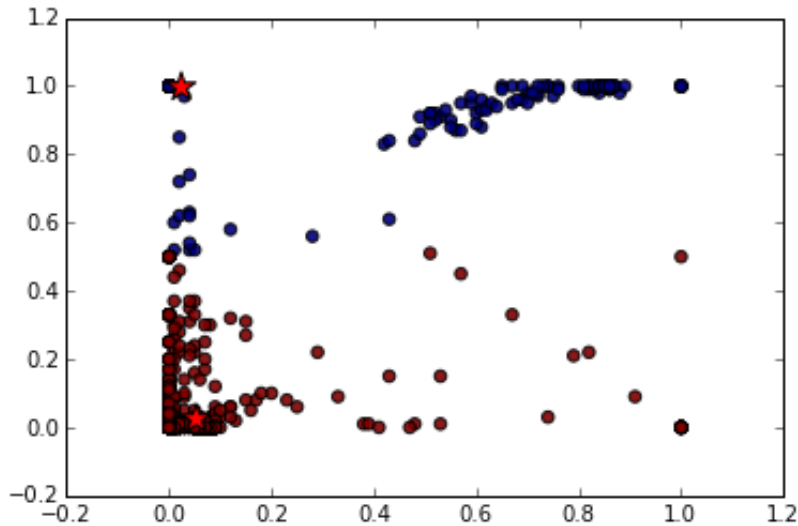


In [372]:

```
oldmu = mu
clusters = cluster_points(X, mu)
mu = reevaluate_centers(oldmu, clusters)
show_clustering(mu, clusters)
```

Out[372]:

<module 'matplotlib.pyplot' from '/usr/local/lib/python2.7/site-packages/matplotlib/pyplot.pyc'>

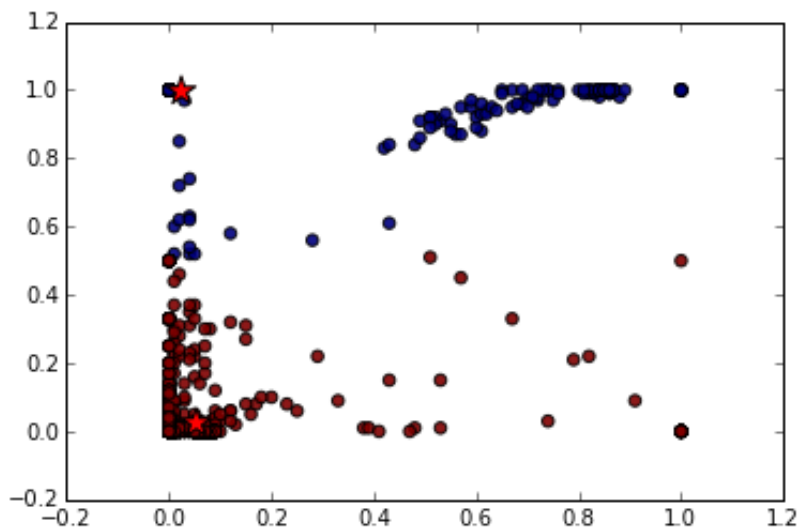


In [373]:

```
oldmu = mu
clusters = cluster_points(X, mu)
mu = reevaluate_centers(oldmu, clusters)
show_clustering(mu, clusters)
```

Out[373]:

<module 'matplotlib.pyplot' from '/usr/local/lib/python2.7/site-packages/matplotlib/pyplot.pyc'>

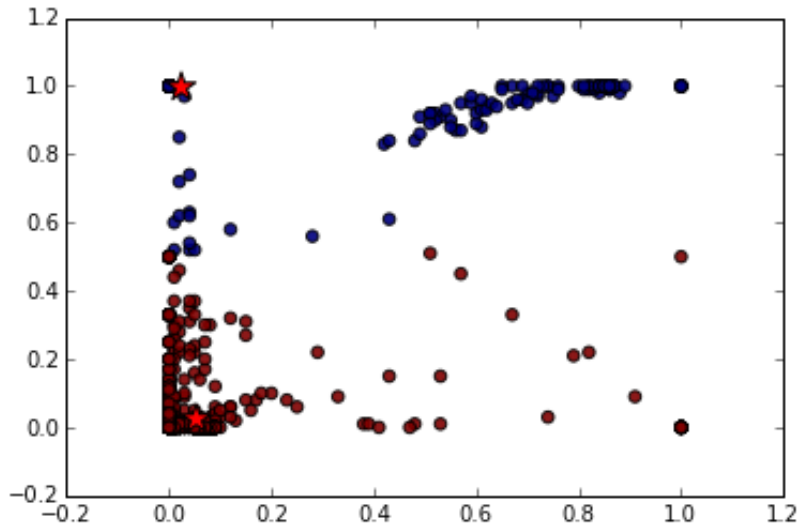


In [374]:

```
oldmu = mu
clusters = cluster_points(X, mu)
mu = reevaluate_centers(oldmu, clusters)
show_clustering(mu, clusters)
```

Out[374]:

<module 'matplotlib.pyplot' from '/usr/local/lib/python2.7/site-packages/matplotlib/pyplot.pyc'>



What would happen if we would use, say, $k = 5$?

In [383]:

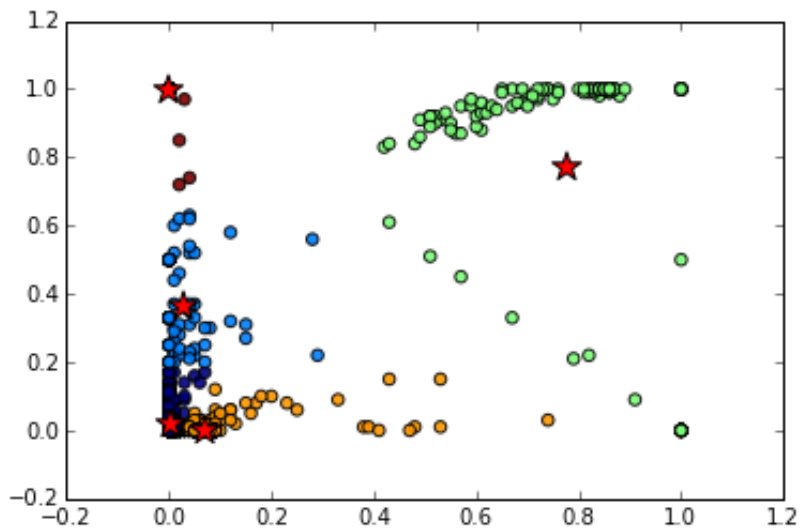
```
K = 5
X = [np.array( (d[33], d[34]) ) for d in data]

mu, clusters = find_centers(X, K)

show_clustering(mu, clusters)
```

Out[383]:

```
<module 'matplotlib.pyplot' from '/usr/local/lib/python2.7/site-packages/matplotlib/pyplot.pyc'>
```



Let's use the clustering procedures provided within spark MLlib.

In [381]:

```
from pyspark.mllib.clustering import KMeans
```

We start by checking how data distributes over the available labels

In [382]:

```
count = raw_data.map(lambda line: line.split(',')[1]).countByValue()  
sorted(list(count.iteritems()), key=lambda e: -e[1])
```

Out[382]:

```
[(u'smurf.', 2744),  
 (u'neptune.', 1027),  
 (u'normal.', 999),  
 (u'satan.', 22),  
 (u'back.', 21),  
 (u'portsweep.', 11),  
 (u'ipsweep.', 8),  
 (u'teardrop.', 6),  
 (u'warezclient.', 6),  
 (u'pod.', 5),  
 (u'nmap.', 4),  
 (u'rootkit.', 1)]
```

The next step consists in converting each data in a vector of pure numbers. We'll simply ignore non-numerical items. We also extract the last item from each line, denoting a label for the corresponding network connection.

In [329]:

```
from pyspark.mllib.linalg import Vectors  
  
def to_example(line):  
    elements = line.split(',')  
    vector = Vectors.dense([float(f) for f in [elements[0]] + elements[4:-1]])  
    label = elements[-1]  
    return (label, vector)  
  
labels_and_data = raw_data.map(to_example)  
data = labels_and_data.values().cache()
```

As a first attempt, let's group using two clusters

In [332]:

```
model = KMeans.train(data, 2)
```

In [333]:

```
model.clusterCenters
```

Out[333]:

```
[array([ 4.92094870e+01,  6.33181189e+02,  5.23683021e+02,
         0.00000000e+00,  4.27524430e-03,  0.00000000e+00,
         4.47882736e-02,  0.00000000e+00,  1.44543974e-01,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         5.29315961e-03,  2.03583062e-04,  2.03583062e-04,
         6.10749186e-04,  0.00000000e+00,  0.00000000e+00,
         1.83224756e-03,  3.37101792e+02,  2.98582858e+02,
         1.70612785e-01,  1.70926303e-01,  5.56066775e-02,
         5.58245114e-02,  7.96182818e-01,  2.00692182e-02,
         2.49063518e-02,  2.33593445e+02,  1.90899023e+02,
         7.61231678e-01,  2.94014658e-02,  6.12962134e-01,
         4.98982085e-03,  1.70934446e-01,  1.70736971e-01,
         5.66021987e-02,  5.55252443e-02]),
 array([ 2.58064516e-01,  5.56316774e+04,  8.01309677e+03,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.90322581e+00,  0.00000000e+00,  9.67741935e-01,
         9.35483871e-01,  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,  3.16129032e+00,  3.35483871e+00,
         0.00000000e+00,  0.00000000e+00,  2.41935484e-02,
         6.45161290e-02,  1.00000000e+00,  0.00000000e+00,
         9.03225806e-02,  1.85000000e+02,  1.82096774e+02,
         9.79354839e-01,  9.67741935e-04,  2.32258065e-02,
         0.00000000e+00,  1.93548387e-03,  1.93548387e-03,
         4.90322581e-02,  4.90322581e-02])]
```

As a first evaluation of the goodness of the obtained clustering, let's check how many points fall in the two clusters, grouping the former according to their label:

In [334]:

```
cluster_label_count = labels_and_data.map(lambda e: (model.predict(e[1]), e[0])
```

In [335]:

```
for (cluster, label), count in sorted(cluster_label_count.iteritems()):
    print '%d\t%17s\t%6d' % (cluster, label, count)
```

```
0          ipsweep.          10
0          neptune.        1043
0          normal.         964
0           pod.             3
0        portsweep.         11
0          satan.           17
0          smurf.        2848
0        teardrop.           6
0       warezclient.         10
1           back.           30
1          normal.           1
```

The result isn't that good: most of items are contained in one of the clusters. Let's try to measure how good a clustering is, computing the average distance of points w.r.t. the assigned centroid's cluster.

In [336]:

```
import math

def distance(a, b):
    return math.sqrt(sum([e**2 for e in a-b]))

def dist_to_centroid(item, model):
    cluster = model.predict(item)
    centroid = model.clusterCenters[cluster]
    return distance(centroid, item)
```

We can use this measure in order to evaluate how many clusters should be used: we start by writing a function building a clustering from data (using a given number of clusters), and returns the average distance measure.

In [337]:

```
def clustering_score(data, k, runs=1, epsilon=1e-4):
    model = KMeans.train(data, k, runs=runs, epsilon=epsilon)
    return data.map(lambda item: dist_to_centroid(item, model)).mean()
```

In [338]:

```
performance = [(k, clustering_score(data, k)) for k in range(5, 40, 5)]
```

In [339]:

```
performance
```

Out[339]:

```
[(5, 586.0273663266862),
 (10, 402.7159081000086),
 (15, 213.19249973499842),
 (20, 154.84384912312305),
 (25, 209.12343610467252),
 (30, 94.83177082412851),
 (35, 80.13452460774573)]
```

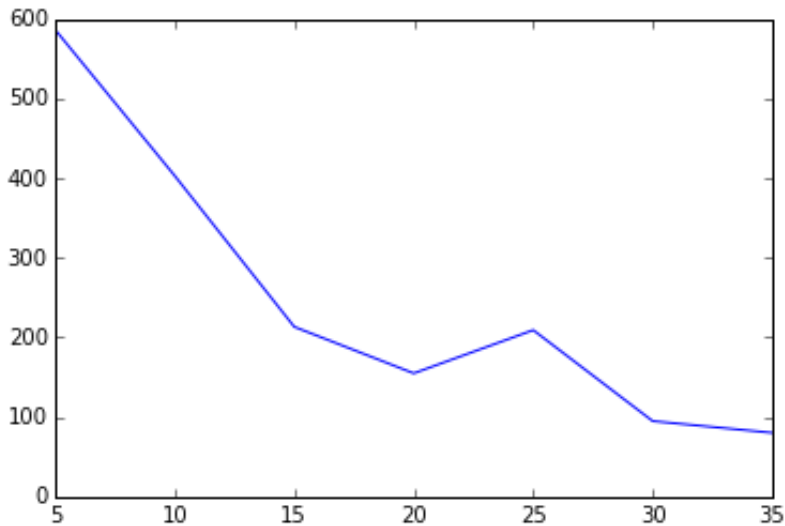
Thus the average distance decreases when the number of clusters increases. That is more than reasonable (should we use a number of clusters as big as the number of points, we would probably have each of the clusters centered on a point, and thus a null average distance). We can exploit this fact, increasing the number k of clusters and stopping when we don't have significant amounts in the decrease of the average distance.

In [340]:

```
%matplotlib inline  
  
import matplotlib.pyplot as plt  
plt.plot(*zip(*performance))
```

Out[340]:

[<matplotlib.lines.Line2D at 0x1143d23d0>]



We typically call such a point in the graph an *ankle*, and in the above picture a good candidate for the ankle could be in correspondence of $k = 30$. We can check this fact by generating again this graph, now using a higher range for k .

In [341]:

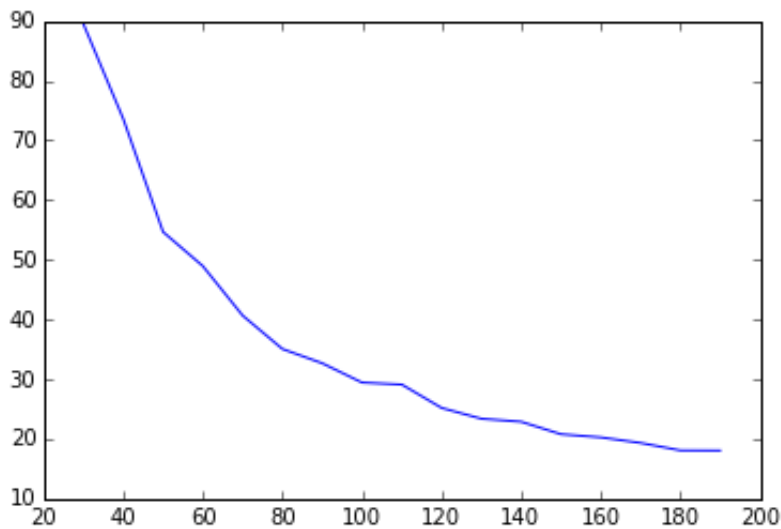
```
performance = [(k, clustering_score(data, k, runs=10, epsilon=1e-6)) for k in
```

In [342]:

```
plt.plot(*zip(*performance))
```

Out[342]:

```
[<matplotlib.lines.Line2D at 0x112d96050>]
```



The graph suggests $k = 130$ as the optimal value. Let's compute the clustering for the optimal value of the number of clusters.

In [125]:

```
model = KMeans.train(data, 130, runs=10, epsilon=1e-6)
```

The points we are clustering belong to a space with dimension 38, thus it is not possible to show them in a graph together with the cluster centroids, thus being able to validate or refute the obtained result. However, we can project each point in a 3D space. We will use a theoretic result showing that, fixed $s \geq 3$ and defining the generic element m_{ij} of the projecting matrix as

$$m_{ij} = \begin{cases} +1 & \text{con probabilità } \frac{1}{2s} \\ 0 & \text{con probabilità } 1 - \frac{1}{s} \\ -1 & \text{con probabilità } \frac{1}{2s} \end{cases}$$

on the average the distances between the projected points will reflect distances in the original space.

In [241]:

```

import random as rnd

def sparse_random_projection(s, n, k):
    '''from Ping Li, Trevor Hastie and Kenneth Church,
    Very Sparse Random Projections, KDD '06

    s: projection parameter (should be geq 3)
    n: original space dimension
    k: reduced space dimension
    ...

    def prob(s):
        u = rnd.random()
        return 1 if u < 1. / (2*s) else -1 if u < 2. / (2*s) else 0

    return np.matrix([[prob(s) for i in range(n)] for j in range(k)])

def project(matrix, point):
    return list(matrix.dot(point).getA()[0])

```

The project function handles the projection of one object in the 3D space, returning the projected point expressed as a list. We are now able to project all points and color them according to the cluster they have been assigned to.

In [247]:

```

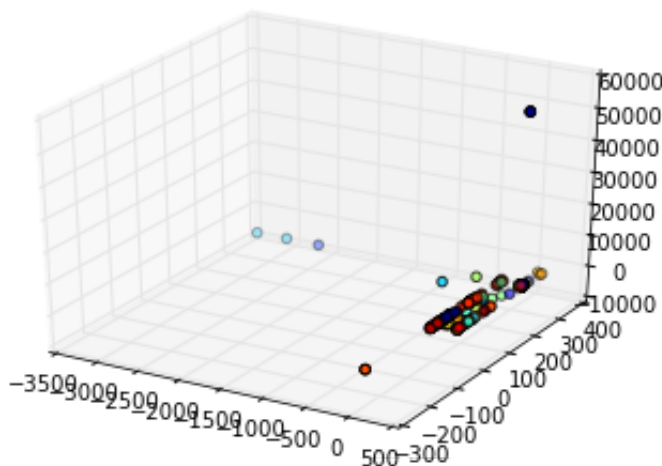
clusters = data.map(lambda item: (np.array(item), model.predict(item))).sample
points, colors = zip(*clusters)

matrix = sparse_random_projection(3, len(points[0]), 3)
xs, ys, zs = zip(*[project(matrix, p) for p in points])

from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xs, ys, zs, c=colors)
plt.show()

```



We see how the variance over one dimension is significantly higher than that on the remaining dimensions. This suggests us that data should be normalized before recomputing the clustering.

In [357]:

```
def build_normalization_function(data):
    data_as_list = data.map(list)
    num_cols = len(data_as_list.first())
    n = data_as_list.count()
    sums = data_as_list.reduce(lambda a, b: [ai+bi for ai, bi in zip(a, b)])
    sumsqares = data_as_list.map(lambda v: [vi**2 for vi in v]).reduce(lambda a, b: [ai+bi for ai, bi in zip(a, b)])
    stdevs = [math.sqrt(n*ss - s*s)/n for ss, s in zip(sumsquares, sums)]
    means = [s/n for s in sums]

    return lambda item: Vectors.dense([0 if s==0 else (i - m)/s for i, m, s in zip(item, means, stdevs)])
```

In [328]:

```
normalized_data = data.map(normalize).cache()
```

In [329]:

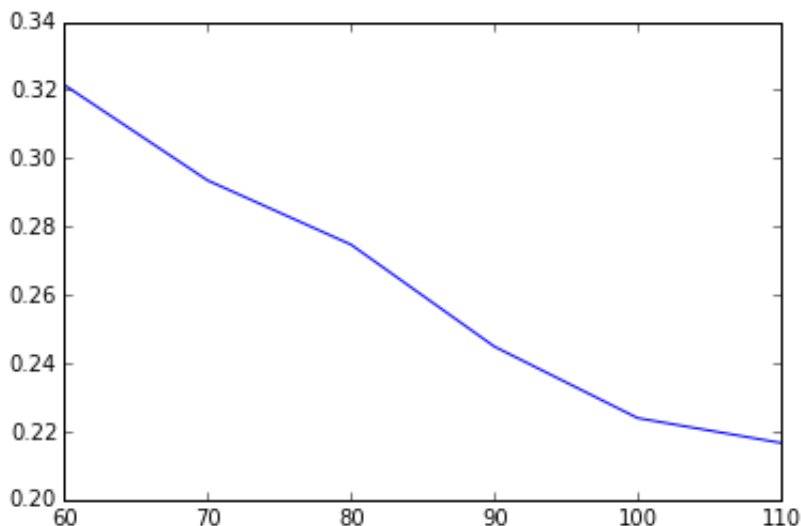
```
performance = [(k, clustering_score(normalized_data, k)) for k in range(60, 120)]
```

In [330]:

```
plt.plot(*zip(*performance))
```

Out[330]:

```
[<matplotlib.lines.Line2D at 0x10bd0ed50>]
```



This graph suggests $k=100$ as an optimal number of clusters. Let's compute the optimal clustering and show the corresponding projected data.

In [331]:

```
model = KMeans.train(normalized_data, 100)
```

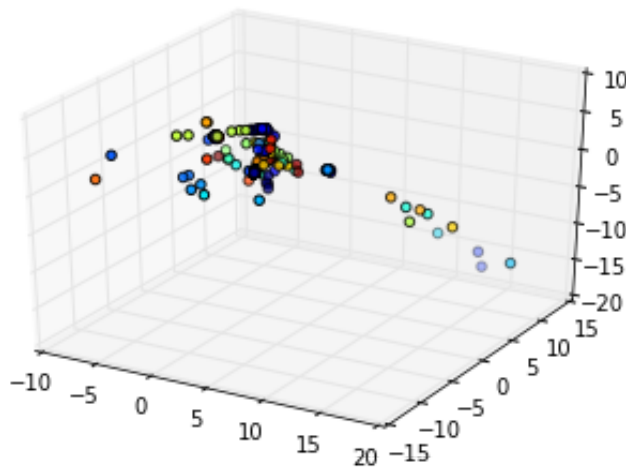
In [332]:

```
clusters = normalized_data.map(lambda item: (np.array(item), model.predict(item)
points, colors = zip(*clusters)

matrix = sparse_random_projection(3, len(points[0]), 3)
xs, ys, zs = zip(*[project(matrix, p) for p in points])

from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xs, ys, zs, c=colors)
plt.show()
```



We can compute the average entropy for points in a same cluster, and use the result as an alternate value of the clustering performance.

In [333]:

```
def entropy(counts):
    values = [c for c in counts if c>0]
    n = float(sum(values))
    return sum([-v/n * math.log(v/n) for v in values])
```

In [358]:

```
from itertools import groupby

def clustering_score_entropy(normalized_labels_and_data, k, runs=1, epsilon=1e-6):
    model = KMeans.train(normalized_labels_and_data.values(), k, runs=runs, epsilon=epsilon)
    labels_and_clusters = normalized_labels_and_data.mapValues(model.predict)
    clusters_and_labels = labels_and_clusters.map(lambda i: list(reversed(i)))
    labels_in_clusters = clusters_and_labels.groupByKey().values()
    label_counts = labels_in_clusters.map(lambda l: [len(list(v)) for k, v in l])
    n = normalized_labels_and_data.count()
    return label_counts.map(lambda m: sum(m) * entropy(m)).sum() / n
```

In [390]:

```
def build_categorical_data_and_label(raw_data):
    split_data = raw_data.map(lambda line: line.split(','))
    protocols_list = split_data.map(lambda line: line[1]).distinct().collect()
    protocols = dict(zip(protocols_list, range(len(protocols_list))))

    services_list = split_data.map(lambda line: line[2]).distinct().collect()
    services = dict(zip(services_list, range(len(services_list))))

    tcp_states_list = split_data.map(lambda line: line[3]).distinct().collect()
    tcp_states = dict(zip(tcp_states_list, range(len(tcp_states_list))))

    def to_categorical_data_and_label(line):
        buf = line.split(',')
        protocol = buf.pop(1)
        service = buf.pop(1)
        tcp_state = buf.pop(1)
        label = buf.pop()
        vector = [float(f) for f in buf]
        new_protocol_features = [0.] * len(protocols)
        new_protocol_features[protocols[protocol]] = 1.0
        new_service_features = [0.] * len(services)
        new_service_features[services[service]] = 1.0
        new_tcp_states_features = [0.] * len(tcp_states)
        new_tcp_states_features[tcp_states[tcp_state]] = 1.0

        vector[1:1] = new_tcp_states_features
        vector[1:1] = new_service_features
        vector[1:1] = new_protocol_features

        return (label, Vectors.dense(vector))
    return to_categorical_data_and_label
```

In [396]:

```
to_categorical_data_and_label = build_categorical_data_and_label(raw_data)
labels_and_data = raw_data.map(to_categorical_data_and_label)
normalize = build_normalization_function(labels_and_data.values())
normalized_labels_and_data = labels_and_data.mapValues(normalize).cache()

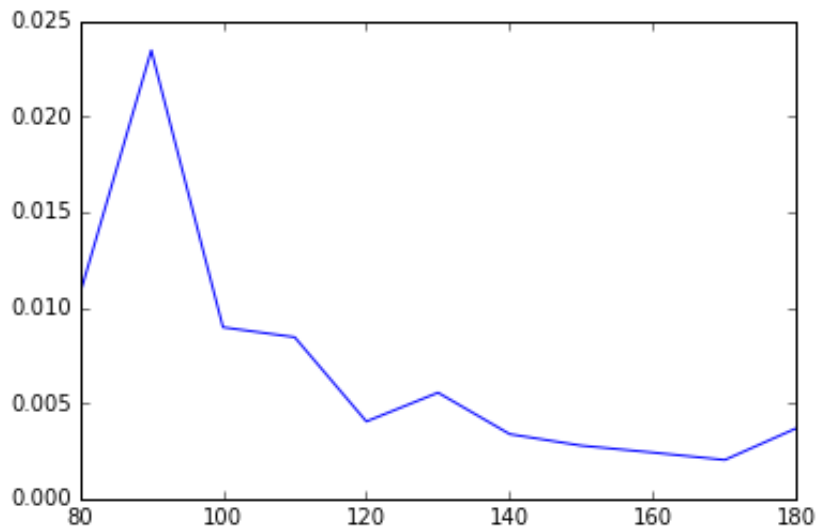
performance = [(k, clustering_score_entropy(normalized_labels_and_data, k)) for
```

In [397]:

```
plt.plot(*zip(*performance))
```

Out[397]:

[<matplotlib.lines.Line2D at 0x10c2045d0>]



Let's use $k=170$ in order to build the clustering, and let's check as in the beginning of the tutorial how many points belong to each cluster, grouping the results with labels.

In [398]:

```
model = KMeans.train(normalized_labels_and_data.values(), 170)
```

In [401]:

```
cluster_label_count = normalized_labels_and_data.map(lambda e: (model.predict(
```

In [402]:

```
for (cluster, label), count in sorted(cluster_label_count.iteritems()):
    print '%d\t%17s\t%6d' % (cluster, label, count)
```

```
0          smurf.          2668
1          satan.           10
2        neptune.         238
3        neptune.           4
4        neptune.           1
5        normal.           1
6        normal.           2
7        neptune.         172
8        normal.           5
9        normal.          84
9         satan.           1
10       neptune.           1
11         back.          15
12       neptune.           1
13        normal.           2
14       neptune.           2
15        normal.         125
16        normal.           1
17       neptune.           1
18 -----
19
```

We can now use this clustering in order to build a detector of anomalous network connections. The basic idea is that of filtering points whose distance from the center of the assigned cluster is higher than a specific threshold value. Such value can be found sorting the distances w.r.t. centers and use the 100-th one as threshold.

In [405]:

```
distances = normalized_labels_and_data.values().map(lambda item: dist_to_centroid(normalize(item), model))
threshold = distances.top(100)[-1]
def is_anomalous(item):
    return dist_to_centroid(normalize(item), model) > threshold
```

We can now filter the connections according to our anomaly detection criterion and show, say, 10 of the filtered ones.

In [409]:

```
normalized_labels_and_data.values().filter(is_anomalous).top(10)
```

Out[409]:

```
[DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,
-0.0348, 0.5443, -0.0269, 2.0588, 0.0, -0.0323, 0.0, -0.0418,
0.0, 2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.019
1, 0.0, 0.0, -0.0318, -1.5428, -1.1715, -0.4664, -0.466, -0.2329,
-0.2327, 0.5347, -0.2454, -0.2, -2.1749, 0.6165, 0.5913, -0.2762,
-1.2412, 0.586, -0.4657, -0.4651, -0.2378, -0.235]),
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,
-0.0348, 0.5443, -0.0262, 0.1252, 0.0, -0.0323, 0.0, -0.0418,
0.0, 2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.019
1, 0.0, 0.0, -0.0318, -1.5569, -1.1837, -0.4664, -0.466, -0.2329,
-0.2327, 0.5347, -0.2454, -0.2, 0.3462, 0.6165, 0.5913, -0.2762,
-1.262, -0.1516, -0.4657, -0.4651, -0.2378, -0.235]),
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,
-0.0348, 0.5443, -0.0265, 0.7223, 0.0, -0.0323, 0.0, -0.0418,
0.0, 2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.019
1, 0.0, 0.0, -0.0318, -1.5711, -1.1959, -0.4664, -0.466, -0.2329,
-0.2327, 0.5347, -0.2454, -0.2, -0.6498, 0.0082, 0.5913, -0.2762,
-1.2412, -0.1516, -0.4657, -0.4651, -0.2378, -0.235]),
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,
-0.0348, 0.5443, -0.0271, 0.388, 0.0, -0.0323, 0.0, -0.0418, 0.0,
2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.0191,
0.0, 0.0, -0.0318, -1.5475, -1.1755, -0.4664, -0.466, -0.2329,
-0.2327, 0.5347, -0.2454, -0.2, -1.7391, -0.657, 0.5913, -0.2762,
-1.2412, -0.1516, -0.4657, -0.4651, -0.2378, -0.235]),
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,
-0.0348, 0.5443, -0.0262, 0.2209, 0.0, -0.0323, 0.0, -0.0418,
0.0, 2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.019
```

```
1, 0.0, 0.0, -0.0318, -1.5475, -1.1755, -0.4664, -0.466, -0.2329,  
-0.2327, 0.5347, -0.2454, -0.2, 0.3462, 0.6165, 0.5913, -0.2762,  
-1.262, -0.1516, -0.4394, -0.4389, -0.2378, -0.235]),  
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,  
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,  
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,  
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,  
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,  
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,  
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,  
-0.0348, 0.5443, -0.026, -0.0127, 0.0, -0.0323, 0.0, -0.0418,  
0.0, 2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.019  
1, 0.0, 0.0, -0.0318, -1.5522, -1.1796, -0.4664, -0.466, -0.2329,  
-0.2327, 0.5347, -0.2454, -0.2, 0.3462, 0.6165, 0.5913, -0.2762,  
-1.262, -0.1516, -0.4657, -0.4651, -0.2378, -0.235]),  
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,  
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,  
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,  
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,  
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,  
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,  
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,  
-0.0348, 0.5443, -0.0275, 10.3602, 0.0, -0.0323, 0.0, -0.0418,  
0.0, 2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.019  
1, 0.0, 0.0, -0.0318, -1.5758, -1.1674, -0.4664, -0.466, -0.2329,  
-0.2327, 0.5347, -0.2454, 1.3775, -2.5328, 0.2838, 0.5913, -0.276  
2, -1.2412, 1.0778, -0.4657, -0.4651, -0.2378, -0.235]),  
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,  
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,  
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,  
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,  
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,  
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,  
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,  
-0.0348, 0.5443, -0.0271, -0.095, 0.0, -0.0323, 0.0, -0.0418,  
0.0, 2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.019  
1, 0.0, 0.0, -0.0318, -1.5758, -1.1999, -0.4664, -0.466, -0.2329,  
-0.2327, 0.5347, -0.2454, -0.2, -3.5443, 0.6165, 0.5913, -0.2762,  
-0.846, 0.586, -0.4657, -0.4651, -0.2378, -0.235]),  
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,  
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,  
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,  
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,  
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,  
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,  
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,  
-0.0348, 0.5443, -0.026, 0.2987, 0.0, -0.0323, 0.0, -0.0418, 0.0,  
2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.0191,  
0.0, 0.0, -0.0318, -1.5099, -1.082, -0.4664, -0.466, -0.2329,  
-0.2327, 0.5347, -0.2454, 0.5171, -3.3265, 0.6165, 0.5913, -0.276  
2, -1.158, 0.586, -0.4657, -0.4651, -0.2378, -0.235]),  
DenseVector([-0.0723, -0.206, -1.1754, 1.2796, -0.0402, -0.0246,  
-0.1554, -0.5372, -0.0201, -0.0492, -0.0913, -0.0201, -0.0142,  
-0.0348, -0.0142, -0.0142, -0.0142, -0.0142, -0.0348, -0.0142,  
-0.0142, -0.0201, -1.1658, -0.0142, -0.0967, -0.0246, -0.0142,  
-0.0142, -0.0318, -0.0201, -0.0142, -0.0142, -0.0201, -0.0142,  
-0.0142, -0.0142, -0.0142, -0.0142, -0.0201, 2.6339, -0.0142,
```

```
-0.0201, -0.1336, -0.0142, -0.0376, -0.4644, -0.2255, -0.0142,  
-0.0348, 0.5443, -0.0261, -0.0164, 0.0, -0.0323, 0.0, -0.0418,  
0.0, 2.3921, -0.062, 0.0, 0.0, -0.0142, -0.0246, -0.0142, -0.019  
1, 0.0, 0.0, -0.0318, -1.491, -1.0494, -0.4664, -0.466, -0.2329,  
-0.2327, 0.5347, -0.2454, 0.1586, -2.3149, 0.6165, 0.5913, -0.276  
2, -1.2412, 0.3402, -0.4657, -0.4651, -0.2378, -0.235]]]
```