Anomaly Detection On IP Address Data

A Simple Example using Gaussian Mixture Modeling (GMM) Clustering

Rajaram Suryanarayanan Follow

Jun 22, 2021 · 12 min read

Photo by Randy Fath on Unsplash

In this article, we will try out unsupervised Machine Learning methods of clustering, for grouping hosts in a IP network in to different clusters based on the similarity in their IP addresses. Importantly, we will see the difference between K-Means and GMM (Gaussian Mixture Model) in their approaches in clustering, and compare their results. We will also see how we can find out the outliers/anomalies in the data using GMM more effectively/

Introduction

Clustering is an unsupervised class of ML algorithms which separate and group given data points in to distinct groups of data in such a way that data points in any cluster are similar to each other in features. Clustering uses the features we supply it to learn the similarities patterns among the population of data points, and then group them accordingly.

There are many different clustering algorithms like K-Means clustering, DBSCAN and GMM to name few popular ones.
In the following exercise, we will extract source and destination IP addresses of packets from a huge data set of network packets and cluster the IPv4 addresses in to different groups based on their four octets, using unsupervised ML method of clustering.
On the way, we will do some anomaly detection and also see how GMM and K-Means compare with each other. Anomalies are outliers (though they are members of the formed clusters) . They are located relatively far away from the rest of the cluster population because of a large degree of dissimilarity, or in other words a very small degree of similarity, with others in the same cluster.

Basics Of K-Means and GMM Clustering Algorithms

We will not go deep in to the details of how K-Means and GMM algorithms works, as that will be too much for the scope of this writing. Instead we will apply both to the task of clustering a population of IP addresses, note how the different approaches taken by the two algorithms have an impact on the resulting cluster formations, and how GMM appear to be better suited to the data set I have in hand.

Some basic information about both algorithms are as follows.

K-Means algorithm loop iterates over two steps, after choosing initial cluster centroids, till the resulting clusters converge.

1. Assign each data observation to the closest cluster centroid
2. Update each centroid to the mean of the data points assigned to it.

So, for K-Means, every data point is assigned to any of the one clusters, this is known as **Hard Clustering** or Hard cluster assignment.

Hard Clustering: In hard clustering, the data points are assigned to any one cluster completely. That means for hard clustering **there is no consideration of uncertainty in the assignment of a data points.**

Soft Clustering:
In soft clustering, the assignment of a data point to a cluster is **based on the probability or likelihood of that data point to existing in that cluster.**

For soft clustering, we have an algorithm called GMMs or Gaussian Mixture Models. **GMM** has two advantages over K-Means:

1. GMM is a lot more flexible regarding cluster covariance and more suited for ellipsoidal shaped blobs.
2. GMM model accommodates mixed membership.

Let's now get in to the details of the example exercise..

Data Pre-Requisite

I took a publicly available pcap file containing millions of packets, extracted packet details like IP header fields, TCP header fields etc and saved it in a large csv file. I used the popular **python tool Scapy** to extract the protocol data of each packet in the pcap file. So, we will start the exercise with a ready-made csv file containing millions of rows with each row describing one network packet.

Code For This Exercise

The full code for this exercise can be followed in this github link.
The notebook is mostly self-explanatory, so I will explain only the important steps and aspects of the exercise in this article. The reader is advised to walk through the full code for the other steps skipped here.

1. Data Preparation

Read the csv file and extract source and destination IP addresses

Read the packets csv file

In [3]:
```
df_file = "pcaps.csv"

df = pd.read_csv(df_file, parse_dates=["timestamp"], \
                 dtype=object, index_col=["timestamp"])

print(sorted(df.columns))
print(df.shape)
```

```
['Unnamed: 0', 'arpdst', 'arpop', 'arppdst', 'arppsrc', 'arpsrc', 'bootpchaddr', 'bootpciaddr', 'bootpgiaddr', 'bootpop', 'boot
psiaddr', 'bootpyiaddr', 'dhcpoptions', 'dnsopcode', 'edst', 'esrc', 'etype', 'icmpcode', 'icmptype', 'id', 'idst', 'iperrords
t', 'iperrorproto', 'iperrorsrc', 'iplen', 'iproto', 'ipttl', 'isrc', 'len', 'ntpmode', 'tdport', 'tsport', 'twindow', 'uerrord
st', 'uerrorsrc', 'ulen', 'utdport', 'utsport', 'vlan']
(5998837, 39)
```

The data has details of each network packet as a row in the data frame.

```
In [5]: df.head(10)
```

Out[5]:

timestamp	Unnamed: 0	id	len	esrc	edst	etype	vlan	isrc	idst	iproto	...	bootpgiaddr	bootpchaddr	dhcpop
2012-03-16 18:00:00	0	0	117	00:16:47:9d:f2:c2	00:0c:29:41:4b:e7	33024	120	192.168.229.254	192.168.202.79	6	...	NaN	NaN	
2012-03-16 18:00:00	1	1	269	00:0c:29:41:4b:e7	00:16:47:9d:f2:c2	33024	120	192.168.202.79	192.168.229.254	6	...	NaN	NaN	
2012-03-16 18:00:00	2	2	70	00:0c:29:41:4b:e7	00:16:47:9d:f2:c2	33024	120	192.168.202.79	192.168.229.251	6	...	NaN	NaN	
2012-03-16 18:00:00	3	3	70	00:16:47:9d:f2:c2	00:0c:29:41:4b:e7	33024	120	192.168.229.254	192.168.202.79	6	...	NaN	NaN	

Above, we have read the csv file containing the packets data which has nearly 6 million rows and 39 columns. Each row contains details of one packet. We are going to use only isrc (Source IP Address) and idst (Destination IP Address) of the packets for our further analysis.

```
In [8]: src_ip_df = df['isrc'].copy()
        dst_ip_df = df['idst'].copy()

        src_ip_df = src_ip_df.reset_index()
        del src_ip_df['timestamp']

        dst_ip_df = dst_ip_df.reset_index()
        del dst_ip_df['timestamp']
```

```
In [9]: print(src_ip_df.shape)
        src_ip_df.head(5)
```

```
(5998837, 1)
```

Out[9]:

	isrc
0	192.168.229.254
1	192.168.202.79
2	192.168.202.79
3	192.168.229.254
4	192.168.202.79

```
In [11]: # Drop missing items from both data frames
         src_ip_df = src_ip_df.dropna().copy()
         dst_ip_df = dst_ip_df.dropna().copy()
```

```
In [12]: # Drop duplicates
         src_ip_df=src_ip_df.drop_duplicates(['isrc'])
         dst_ip_df=dst_ip_df.drop_duplicates(['idst'])
         print(src_ip_df.shape)
         print(dst_ip_df.shape)

         (187, 1)
         (2673, 1)
```

After data cleaning and dropping duplicate entries, what **we have are 187 Source IP addresses and 2673 Destination IP addresses for analysis.**

2. Features extraction from IP addresses

For clustering of data, we need to choose particular features of the data based on which the clustering algorithm can divide the data in to clusters of similar members. Here, since we want the clustering to be based on IPv4 addresses, we need some way to encode the IP address in to a vector of features.

We will use the four octets of IP address as features for clustering.

```
In [13]: src_ip_df.loc[:, 'oct1'] = src_ip_df['isrc'].apply(lambda x: x.split(".")[0])
         src_ip_df.loc[:, 'oct2'] = src_ip_df['isrc'].apply(lambda x: x.split(".")[1])
         src_ip_df.loc[:, 'oct3'] = src_ip_df['isrc'].apply(lambda x: x.split(".")[2])
         src_ip_df.loc[:, 'oct4'] = src_ip_df['isrc'].apply(lambda x: x.split(".")[3])

         dst_ip_df.loc[:, 'oct1'] = dst_ip_df['idst'].apply(lambda x: x.split(".")[0])
         dst_ip_df.loc[:, 'oct2'] = dst_ip_df['idst'].apply(lambda x: x.split(".")[1])
         dst_ip_df.loc[:, 'oct3'] = dst_ip_df['idst'].apply(lambda x: x.split(".")[2])
         dst_ip_df.loc[:, 'oct4'] = dst_ip_df['idst'].apply(lambda x: x.split(".")[3])
```

```
In [17]: src_ip_df = src_ip_df.reset_index()
         print(src_ip_df.head(5))
         dst_ip_df = dst_ip_df.reset_index()
         print(src_ip_df.head(5))
```

```
   index            isrc oct1 oct2 oct3 oct4
0      0  192.168.229.254  192  168  229  254
1      1   192.168.202.79  192  168  202   79
2      2  192.168.229.251  192  168  229  251
3      3  192.168.229.153  192  168  229  153
4      4    192.168.215.1  192  168  215    1
   index            isrc oct1 oct2 oct3 oct4
0      0  192.168.229.254  192  168  229  254
1      1   192.168.202.79  192  168  202   79
2      2  192.168.229.251  192  168  229  251
3      3  192.168.229.153  192  168  229  153
4      4    192.168.215.1  192  168  215    1
```

We have chosen to split the four octets of each IPv4 address and use them as a vector of length 4 to be fed to the clustering algorithm to represent each data (IP address). The reasoning behind choosing this approach was taken from this research paper.

```
In [18]: X_matrix_src = np.array(src_ip_df[['oct1', 'oct2', 'oct3', 'oct4']])
         print(X_matrix_src.shape)
         print(X_matrix_src[0:5])
         print()

         X_matrix_dst = np.array(dst_ip_df[['oct1', 'oct2', 'oct3', 'oct4']])
         print(X_matrix_dst.shape)
         print(X_matrix_dst[0:5])
```

```
(187, 4)
[['192' '168' '229' '254']
 ['192' '168' '202' '79']
 ['192' '168' '229' '251']
 ['192' '168' '229' '153']
 ['192' '168' '215' '1']]
(2673, 4)
[['192' '168' '202' '79']
 ['192' '168' '229' '254']
 ['192' '168' '229' '251']
 ['192' '168' '229' '153']
 ['224' '0' '0' '10']]
```

We then apply PCA to the X_matrix to reduce the dimensions from 4 to 2, so that we can visualize the clusters by scatter plot easily with 2 dimensions.

```
In [22]: src_ip_df['pca1'] = pcas1
         src_ip_df['pca2'] = pcas2
         print(src_ip_df[:2])
         print()
         dst_ip_df['pca1'] = pcad1
         dst_ip_df['pca2'] = pcad2

         print(dst_ip_df[:2])
```

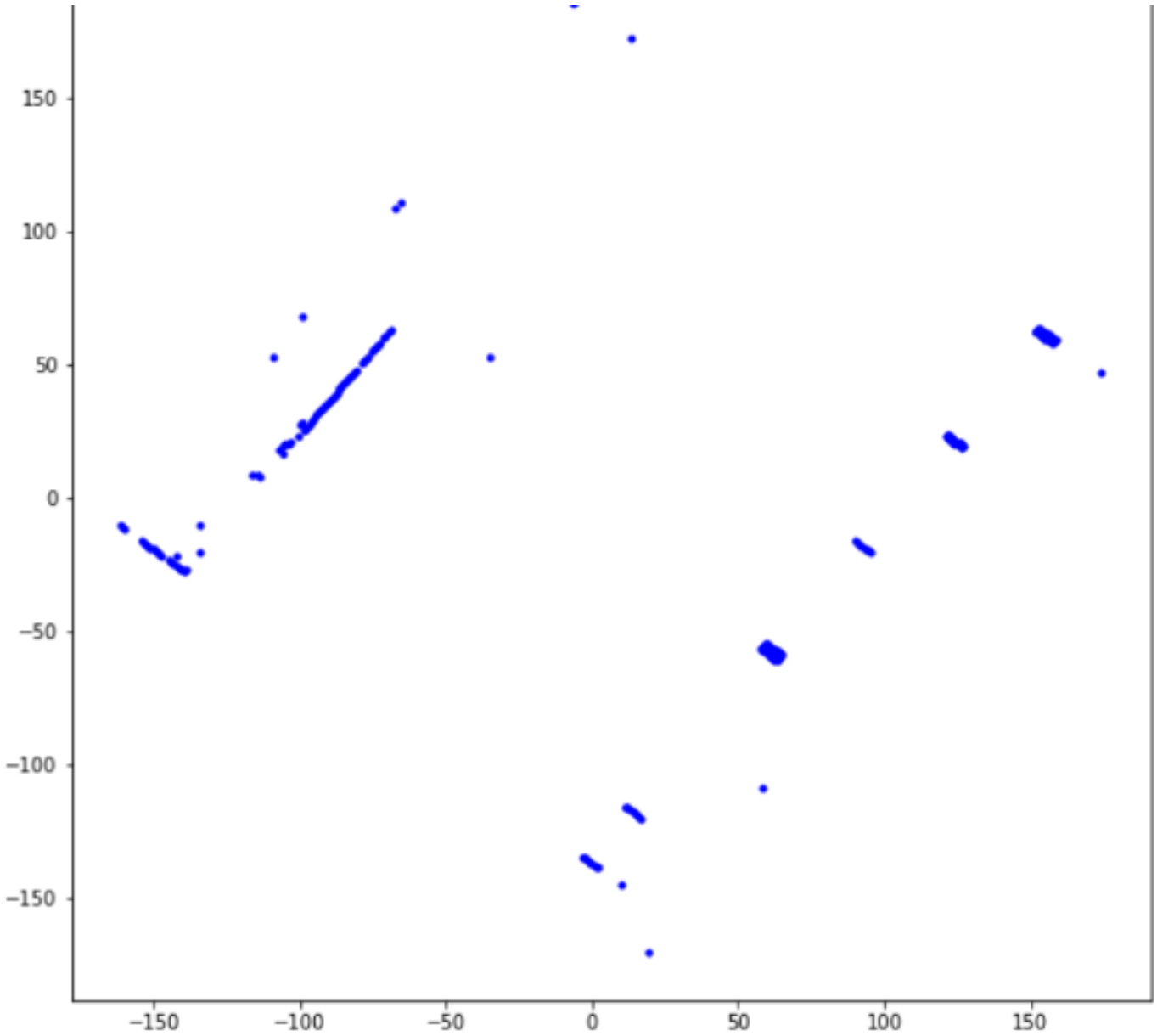
	index	isrc	oct1	oct2	oct3	oct4	pca1	pca2
0	0	192.168.229.254	192	168	229	254	-4.762	187.744
1	1	192.168.202.79	192	168	202	79	-91.854	33.929

	index	idst	oct1	oct2	oct3	oct4	pca1	pca2
0	0	192.168.202.79	192	168	202	79	100.201	110.117
1	1	192.168.229.254	192	168	229	254	-46.441	209.185

When we scatter plot the pca1 vs pca2 of src_ip_df , we get the following plot

Scatter plot of Source IP address data

```
In [24]: import matplotlib.pyplot as plt
         %matplotlib inline
         plt.rcParams["figure.figsize"] = (10,10)
         plt.scatter(src_ip_df['pca1'], src_ip_df['pca2'], s=10, color='blue',label="All Hosts")
         plt.legend(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)
         plt.show()
```



Visually, it seems like 6 or 7 small clusters are there. Broadly there are two big clusters with ellipsoidal shape and similar orientation. And the cluster patterns are more ellipsoidal, particularly the ones on the left.

3. Use Gaussian Mixture Modeling (GMM) Clustering algorithm On Source IP Address data

Gaussian Mixture Model (GMM) clustering suits ellipsoidal shaped clusters more than K-Means clustering which suits spherical blobs. GMM also is a probabilistic clustering algorithm and provides easier way to detect anomalies. Let us try GMM to cluster the source IP addresses.

We use BayesianGaussianMixture class which helps us finding the optimum number of clusters needed for the given data. If we input a number of clusters that is more than what is optimum, it eliminates unnecessary clusters by giving weights equal to or close to zero to those unnecessary clusters.

Use BayesianGaussianMixture to determine first the optimum number of clusters that can be formed.

In [25]:
```
from sklearn.mixture import BayesianGaussianMixture

# Check 7 clusters
bgms = BayesianGaussianMixture(n_components=7, n_init=10, random_state=100)

bgms.fit(X_matrix_src)
np.round(bgms.weights_, 2)
```

Out[25]:
```
array([0.47, 0.52, 0.01, 0.  , 0.  , 0.  , 0.  ])
```

BayesianGaussianMixture gives weights of each cluster. We see that few of the clusters have been eliminated with 0 weight. So we can try 3 clusters on the data.

In [26]:
```
bgms = BayesianGaussianMixture(n_components=3, n_init=10, random_state=100)
bgms.fit(X_matrix_src)
print(np.round(bgms.weights_, 2))
```

```
[0.53 0.46 0.01]
```

We tried 7 clusters, but 3 seems enough, going by the weights given to the clusters. We then train BGM (Bayesian Gaussian Mixture) on source IP data asking for three clusters to be created. In the output seen above, we see the weights assigned by BGM for each of the three clusters created by the algorithm.

We can also get the Gaussian means of the three clusters and transform it via PCA as seen below, for locating the means of the clusters during visualization.

Observe the means of the clusters, apply PCA and save it in data frame for plotting

In [27]:
```
print(bgms.means_)
pca_means = pca_src.transform(bgms.means_)
print()
print(pca_means)
```

```
[[191.98854859 167.9828769   24.97850159 144.93134559]
 [191.98711716 167.98073651 205.70308719  70.74094526]
 [120.95546619  61.76831501  36.95726324  58.40109023]]

[[ 87.99445487 -23.61248302]
 [-99.87397803  29.7390786 ]
 [ 25.8196479  -93.134321  ]]
```

In [28]:
```
means_df_s = pd.DataFrame(pca_means, columns=['pca1','pca2'])
means_df_s
```

Out[28]:

	pca1	pca2
0	87.994	-23.612
1	-99.874	29.739
2	25.820	-93.134

Next we predict the cluster for all the source IP data

In [30]:
```
bgms.predict(X_matrix_src)
```

Out[30]:
```
array([1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0,
       1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0,
       1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 2, 1, 2, 0, 0, 1, 0, 1, 1, 1, 1,
       0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
       0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 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, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
       0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0], dtype=int64)
```

In [31]:
```
src_ip_df['kcluster']=bgms.predict(X_matrix_src).tolist()
src_ip_df.tail(10)
```

Out[31]:

	index	isrc	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
177	177	192.168.202.86	192	168	202	86	-87.522	39.415	1
178	178	192.168.202.113	192	168	202	113	-70.813	60.577	1
179	179	192.168.201.2	192	168	201	2	-138.721	-27.037	1
180	180	192.168.202.82	192	168	202	82	-89.998	36.280	1
181	181	192.168.202.115	192	168	202	115	-69.575	62.144	1

As seen above, we store the cluster label for each IP address as a separate column ‘kcluster’. Also we create separate data frames for each cluster so that it will help in plotting the clusters.

Plot the clusters identified by GMM

```
In [32]: dfs0=src_ip_df[src_ip_df.kcluster==0]
         dfs1=src_ip_df[src_ip_df.kcluster==1]
         dfs2=src_ip_df[src_ip_df.kcluster==2]

In [33]: plt.rcParams["figure.figsize"] = (16,8)
         pyplot.scatter(dfs0['pca1'],dfs0['pca2'],s=20,label="cl 0")
         pyplot.scatter(dfs1['pca1'],dfs1['pca2'],s=20,marker="<",label="cl 1")
         pyplot.scatter(dfs2['pca1'],dfs2['pca2'],s=20,marker="D",label="cl 2")
         pyplot.scatter(means_df_s['pca1'],means_df_s['pca2'],s=50,marker="X",label="Mean", color='black')

         pyplot.legend(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)
         pyplot.show()
```

We plot the three clusters created by GMM as the below image shows.

We can see 3 clusters formed neatly, but do contain some outliers/anomalies.

On analyzing the members of each cluster, we see that with GMM clustering, 3 clusters were clearly formed from the source IP addreses.

```
In [35]: dfs0
```

Out[35]:

	index	isrc	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
6	6	192.168.27.25	192	168	27	25	12.186	-116.361	0
11	11	192.168.27.103	192	168	27	103	60.457	-55.228	0
13	13	192.168.27.253	192	168	27	253	153.286	62.336	0
15	15	192.168.27.152	192	168	27	152	90.781	-16.824	0
16	16	192.168.27.102	192	168	27	102	59.838	-56.012	0

...	...	...	...	...	...	...	...	...	...
174	174	192.168.27.252	192	168	27	252	152.667	61.552	0
175	175	192.168.24.254	192	168	24	254	156.261	61.269	0

Cluster 0 consists of mostly 192.168.(21–27) subnets with a couple of possible outliers / anomalies.

```
In [37]: dfs1
```

Out[37]:

	index	isrc	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
0	0	192.168.229.254	192	168	229	254	-4.762	187.744	1
1	1	192.168.202.79	192	168	202	79	-91.854	33.929	1
2	2	192.168.229.251	192	168	229	251	-6.618	185.393	1
3	3	192.168.229.153	192	168	229	153	-67.267	108.585	1
4	4	192.168.215.1	192	168	215	1	-150.336	-19.183	1
...	...	...	...	...	...	...	...	...	...
180	180	192.168.202.82	192	168	202	82	-89.998	36.280	1
181	181	192.168.202.115	192	168	202	115	-69.575	62.144	1

Cluster 1 consists of mostly 192.168.(200+) subnets. This cluster is not so tight, have a sparsely populated range of third octet which seem to have possible anomalies.

```
In [39]: dfs2
```

Out[39]:

	index	isrc	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
77	77	172.19.2.66	172	19	2	66	58.107	-108.918	2
79	79	0.0.0.0	0	0	0	0	19.352	-170.485	2

Cluster 2 consists of only two IP addresses which clearly seem to be anomalies, when compared to the cluster 0 and 1 addresses.

4. Find Anomalies/Outliers from the clusters of IP addresses

We use score_samples() method of GMM to estimate the density at the location of each data point in the clusters. The greater the score, higher the density of the cluster at the location of that data point. Any instance located in a low-density region can be considered an anomaly. We can define a density threshold of say 4%, and consider all the data lying in areas of density below 4th percentile of the range of densities values, as anomalies. **This threshold is arbitrary and we can choose it according to our discretion.**

In [41]:
```
densities_s = bgms.score_samples(X_matrix_src)
dens_threshold_s = np.percentile(densities_s, 4)

print("Maximum Density : " + str(max(densities_s)))
print("Minimum Density : " + str(min(densities_s)))
print("Threshold Density : " + str(dens_threshold_s))

anomalies_s = X_matrix_src[densities_s < dens_threshold_s]
```

```
Maximum Density : -11.657736542696394
Minimum Density : -27.048463652842244
Threshold Density : -14.049126105279296
```

In [42]:
```
print(anomalies_s)
print(len(anomalies_s))
```

```
[['192' '168' '229' '254']
 ['192' '168' '229' '251']
 ['192' '168' '205' '253']
 ['192' '168' '1' '254']
 ['192' '168' '229' '252']
 ['172' '19' '2' '66']
 ['192' '168' '169' '129']
 ['0' '0' '0' '0']]
8
```

In [43]:
```
anomalies_df_s = src_ip_df[densities_s < dens_threshold_s]
anomalies_df_s
```

Out[43]:

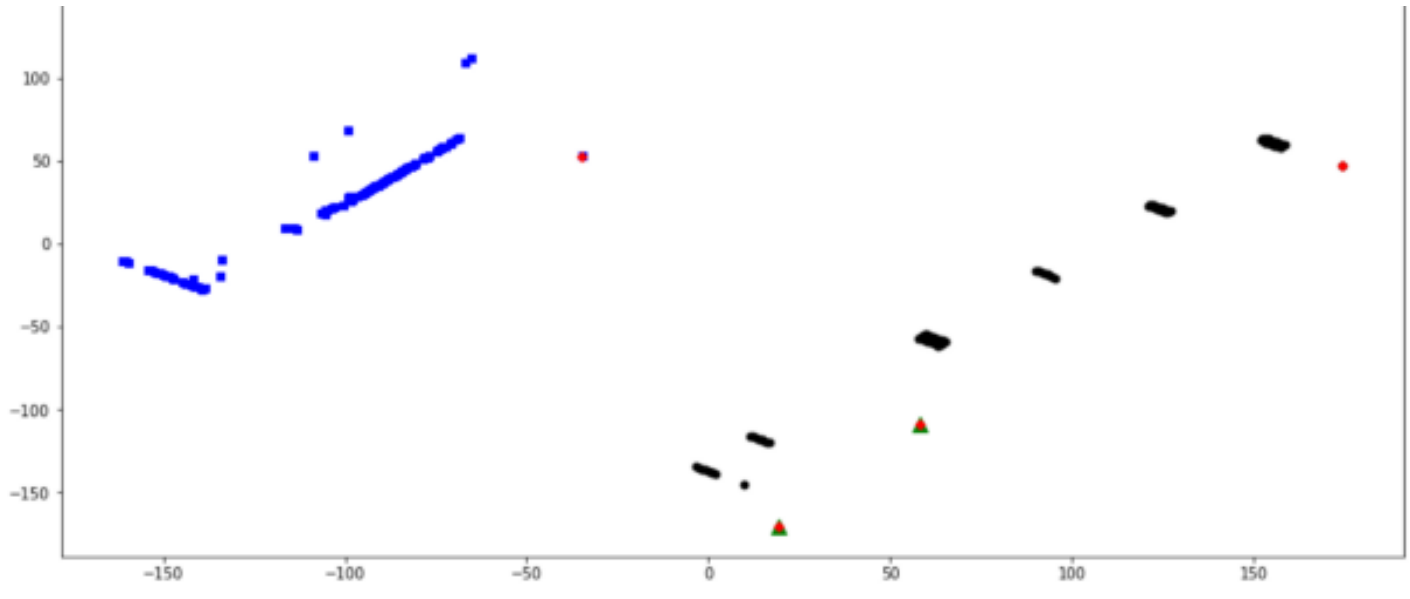
	index	isrc	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
0	0	192.168.229.254	192	168	229	254	-4.762	187.744	1
2	2	192.168.229.251	192	168	229	251	-6.618	185.393	1
45	45	192.168.205.253	192	168	205	253	13.471	172.154	1
46	46	192.168.1.254	192	168	1	254	174.327	47.079	0
56	56	192.168.229.252	192	168	229	252	-6.000	186.177	1
77	77	172.19.2.66	172	19	2	66	58.107	-108.918	2
78	78	192.168.169.129	192	168	169	129	-34.991	52.758	1
79	79	0.0.0.0	0	0	0	0	19.352	-170.485	2

We have identified eight IP addresses as anomalies in the set of source IP addresses.

We plot the clusters again, this time marking anomaly data points.

Plot the clusters again, this time with anomalies

In [44]:
```
plt.rcParams["figure.figsize"] = (16,8)
pyplot.scatter(dfs0['pca1'],dfs0['pca2'],s=25,label="cl 0", color='black')
pyplot.scatter(dfs1['pca1'],dfs1['pca2'],s=25,marker=",",label="cl 1", color='blue')
pyplot.scatter(dfs2['pca1'],dfs2['pca2'],s=100,marker="^",label="cl 2", color='green')
pyplot.scatter(anomalies_df_s['pca1'],anomalies_df_s['pca2'],s=20,marker="D",label="Anomalies", color='red')

pyplot.legend(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)
pyplot.show()
```



The cluster diagram clearly shows anomalies (in red) lying far away from the tight cluster groups. Also if you compare the anomalies IP addresses with those commonly found in the clusters, you can see striking differences.

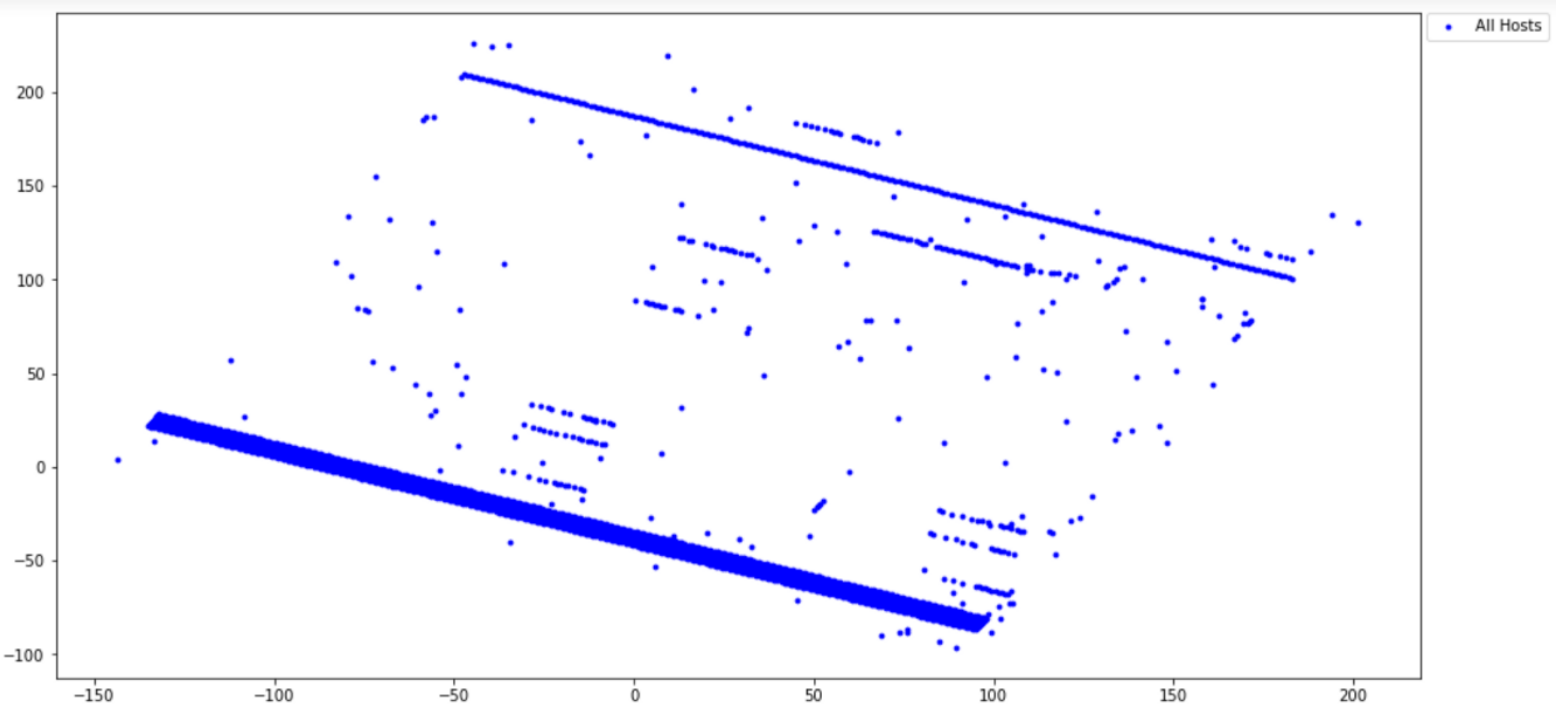
For example, many anomalies end with last octet in range >250 (i.e towards the end of the octet range 0–255). Both addresses of cluster 2 -172.19.2.66 and 0.0.0.0 are marked anomalies. This is expected looking their values which stand out from 192.168. IP addresses in cluster 0 and 1.
It is the same case with 192.168.169.129 whose third octet 169 stands apart from the observed range.

So, we have successfully used Gaussian Mixture Modeling to cluster the source IP addresses in the packet data and also extracted the possible anomalies/outliers in the addresses used.

5. Repeat the same exercise with destination IP addresses and see how GMM fares.

Here, we repeat the same steps for destination IP addresses, that we already followed for source IP addresses.

First we plot the IP address data using the two corresponding PCA components.



The destination IP addresses clearly seem to be more ellipsoidal with at least two long clusters at the top and bottom with similar shape and orientations.

Below we estimate the optimum number of clusters to be 4 and train accordingly with the destination IP data.

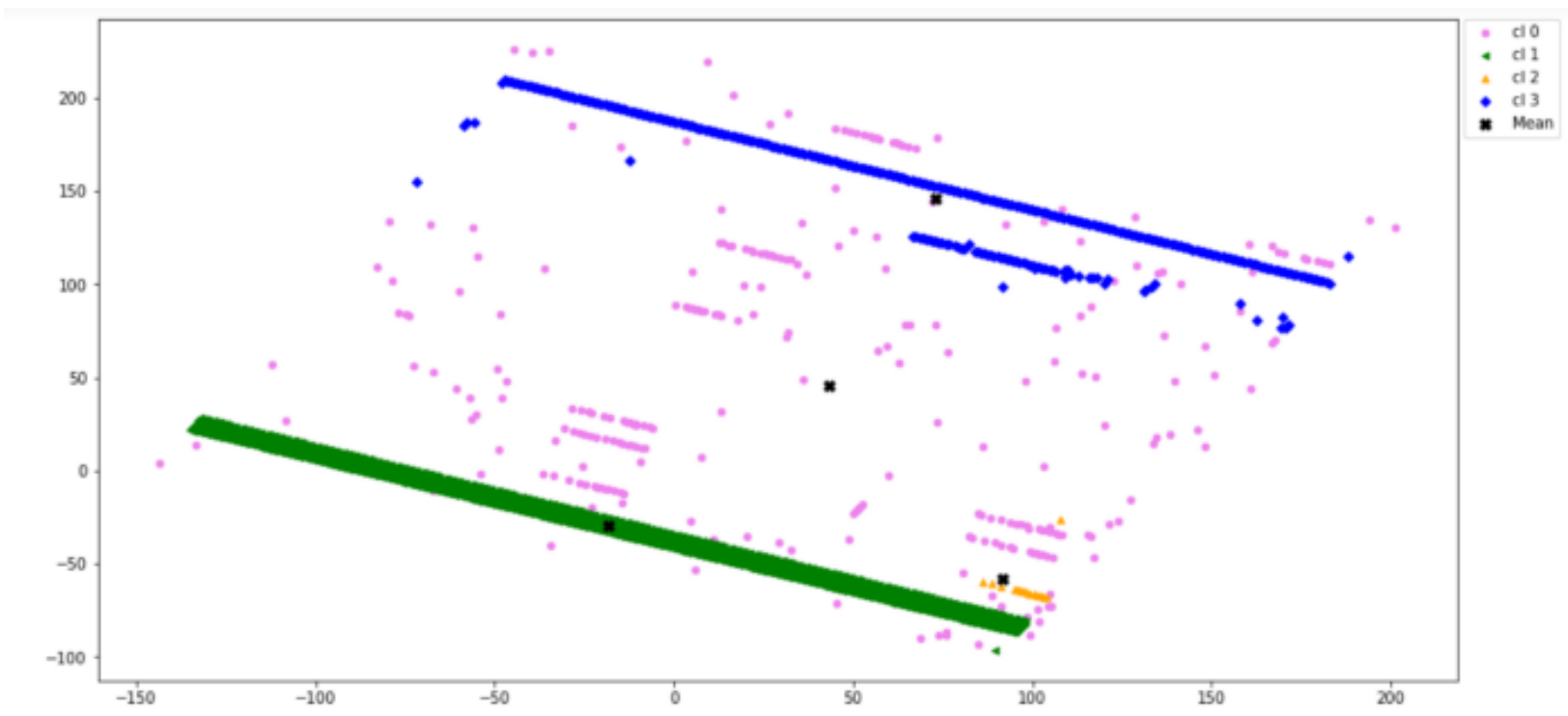
```
In [47]: from sklearn.mixture import BayesianGaussianMixture
         bgmd = BayesianGaussianMixture(n_components=5, n_init=10, random_state=100)

         bgmd.fit(X_matrix_dst)
         np.round(bgmd.weights_, 2)

Out[47]: array([0.77, 0.01, 0.1 , 0.  , 0.12])
```

BayesianGaussianMixture has eliminated one cluster and it looks like 4 clusters will be optimum

```
In [48]: bgmd = BayesianGaussianMixture(n_components=4, n_init=10, random_state=100)
         bgmd.fit(X_matrix_dst)
         np.round(bgmd.weights_, 2)

Out[48]: array([0.1 , 0.77, 0.01, 0.12])
```

Similar to what we saw with source IP data in the preceding sections, we can predict the clusters for each destination IP address and plot the same as below.

```
In [55]: plt.rcParams["figure.figsize"] = (16,8)
         pyplot.scatter(dfd0['pca1'],dfd0['pca2'],s=20,label="cl 0", color='violet')
         pyplot.scatter(dfd1['pca1'],dfd1['pca2'],s=20,marker="<",label="cl 1", color='green')
         pyplot.scatter(dfd2['pca1'],dfd2['pca2'],s=20,marker="^",label="cl 2", color='orange')
         pyplot.scatter(dfd3['pca1'],dfd3['pca2'],s=20,marker="D",label="cl 3", color='blue')
         pyplot.scatter(means_df_d['pca1'],means_df_d['pca2'],s=50,marker="X",label="Mean", color='black')

         pyplot.legend(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)
         pyplot.show()
```



We see GMM has decently clustered the IP addresses in to three large clusters and one small cluster (cl 2).

Important to note that **GMM has respected the ellipsoidal shapes and orientation of the clusters well. This is what differentiates GMM from K-Means which assumes clusters to be spherical (which we will see subsequently below).**

Then we analyze each cluster to see how similar the IP addresses are in each.

In [57]:
```
# Cluster 0
print(dfd0.shape)
dfd0.head(20)
```

```
(278, 9)
```

Out[57]:

	index	idst	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
4	4	224.0.0.10	224	0	0	10	84.847	-93.207	0
10	10	64.4.23.149	64	4	23	149	-22.078	-8.396	0
14	14	157.56.52.15	157	56	52	15	102.960	-45.208	0
18	18	157.55.56.150	157	55	56	150	-17.210	16.091	0
19	19	157.55.235.162	157	55	235	162	47.493	182.681	0
21	21	178.63.97.34	178	63	97	34	103.203	2.458	0
24	24	149.5.45.166	149	5	45	166	-33.046	16.025	0

Cluster 0 appears very scattered with addresses from a wide range of subnets.
We can also see this from the clusters plot above. **Looks like it will contain a lot of anomalies.**

In [59]:
```
# Cluster 1
print(dfd1.shape)
dfd1.head(20)
```

```
(2049, 9)
```

Out[59]:

	index	idst	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
5	5	192.168.27.25	192	168	27	25	75.115	-70.795	1
7	7	192.168.27.253	192	168	27	253	-130.784	26.545	1
9	9	192.168.27.152	192	168	27	152	-39.574	-16.575	1
11	11	192.168.27.102	192	168	27	102	5.579	-37.921	1
12	12	192.168.27.103	192	168	27	103	4.676	-37.494	1
13	13	192.168.27.101	192	168	27	101	6.482	-38.348	1
15	15	192.168.27.1	192	168	27	1	96.788	-81.041	1
16	16	192.168.27.203	192	168	27	203	-85.631	5.198	1
17	17	192.168.27.202	192	168	27	202	-84.727	4.772	1
27	27	192.168.21.203	192	168	21	203	-88.163	-0.214	1

In [60]:
```
print(dfd1['oct1'].value_counts())
print(dfd1['oct2'].value_counts())
print(dfd1['oct3'].value_counts())
```

```
192    2049
Name: oct1, dtype: int64
168    2049
Name: oct2, dtype: int64
28     256
25     256
```

```
27    256
24    256
23    256
26    256
21    256
22    256
10      1
Name: oct3, dtype: int64
```

Cluster 1 is very tight containing almost only 192.168.[21–28] subnet addresses. This cluster is the most populated from all the four clusters. Only one outlier — 192.168.10.* we can see sitting far in the cluster plot too.

```
In [61]: # Cluster 2
         print(dfd2.shape)
         dfd2.head(20)
```

```
(15, 9)
```

Out[61]:

	index	idst	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
185	185	183.182.82.14	183	182	82	14	107.989	-26.379	2
227	227	173.194.43.4	173	194	43	4	100.459	-66.186	2
230	230	173.194.43.20	173	194	43	20	86.010	-59.355	2
245	245	173.194.43.10	173	194	43	10	95.041	-63.624	2
2640	2640	173.194.43.1	173	194	43	1	103.168	-67.466	2
2641	2641	173.194.43.8	173	194	43	8	96.847	-64.478	2
2642	2642	173.194.43.7	173	194	43	7	97.750	-64.905	2
2643	2643	173.194.43.14	173	194	43	14	91.428	-61.916	2

```
In [62]: print(dfd2['oct1'].value_counts())
         print(dfd2['oct2'].value_counts())
         print(dfd2['oct3'].value_counts())
```

```
173    14
183     1
Name: oct1, dtype: int64
194    14
182     1
Name: oct2, dtype: int64
43    14
82     1
Name: oct3, dtype: int64
```

Cluster 2 is small, very tight containing 173.194.43 subnet addresses. This cluster is the least populated from all the four clusters. One outlier 183.182.82.14 is seen which is

also seen standing out in the cluster plot above.

In [63]:
```
# Cluster 3
print(dfd3.shape)
dfd3.head(20)
```

```
(331, 9)
```

Out[63]:

	index	idst	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
0	0	192.168.202.79	192	168	202	79	100.201	110.117	3
1	1	192.168.229.254	192	168	229	254	-46.441	209.185	3
2	2	192.168.229.251	192	168	229	251	-43.731	207.905	3
3	3	192.168.229.153	192	168	229	153	44.769	166.065	3
6	6	192.168.202.100	192	168	202	100	81.237	119.083	3
8	8	192.168.169.254	192	168	169	254	-71.761	155.063	3
20	20	192.168.202.76	192	168	202	76	102.911	108.837	3
22	22	192.168.202.89	192	168	202	89	91.171	114.387	3
23	23	192.168.229.101	192	168	229	101	91.728	143.865	3
25	25	192.168.202.255	192	168	202	255	-58.738	185.257	3

In [64]:
```
print(dfd3['oct1'].value_counts())
print(dfd3['oct2'].value_counts())
print(dfd3['oct3'].value_counts())
```

```
192    330
194      1
Name: oct1, dtype: int64
168    330
165      1
Name: oct2, dtype: int64
229    256
202     50
204      7
203      4
205      3
201      2
208      2
188      1
207      1
227      1
206      1
200      1
169      1
244      1
Name: oct3, dtype: int64
```

Cluster 3 is tight containing mostly 192.168.[>200] subnet addresses.

So with GMM clustering, we saw the above four clusters clearly formed from the destination IP addresses.

We then separate out anomalies from the data, using the same densities threshold comparison, that we did for source IP address analysis.

```
In [67]: densities_d = bgmd.score_samples(X_matrix_dst)
         dens_threshold_d = np.percentile(densities_d, 4)
         print("Maximum Density: " + str(max(densities_d)))
         print("Minimum Density: " + str(min(densities_d)))
         print("Threshold Density: " + str(dens_threshold_d))

         anomalies_d = X_matrix_dst[densities_d < dens_threshold_d]
```

```
Maximum Density: -8.40416989144428
Minimum Density: -28.704807017163787
Threshold Density: -25.026364347639266
```

```
In [68]: print(len(anomalies_d))
```

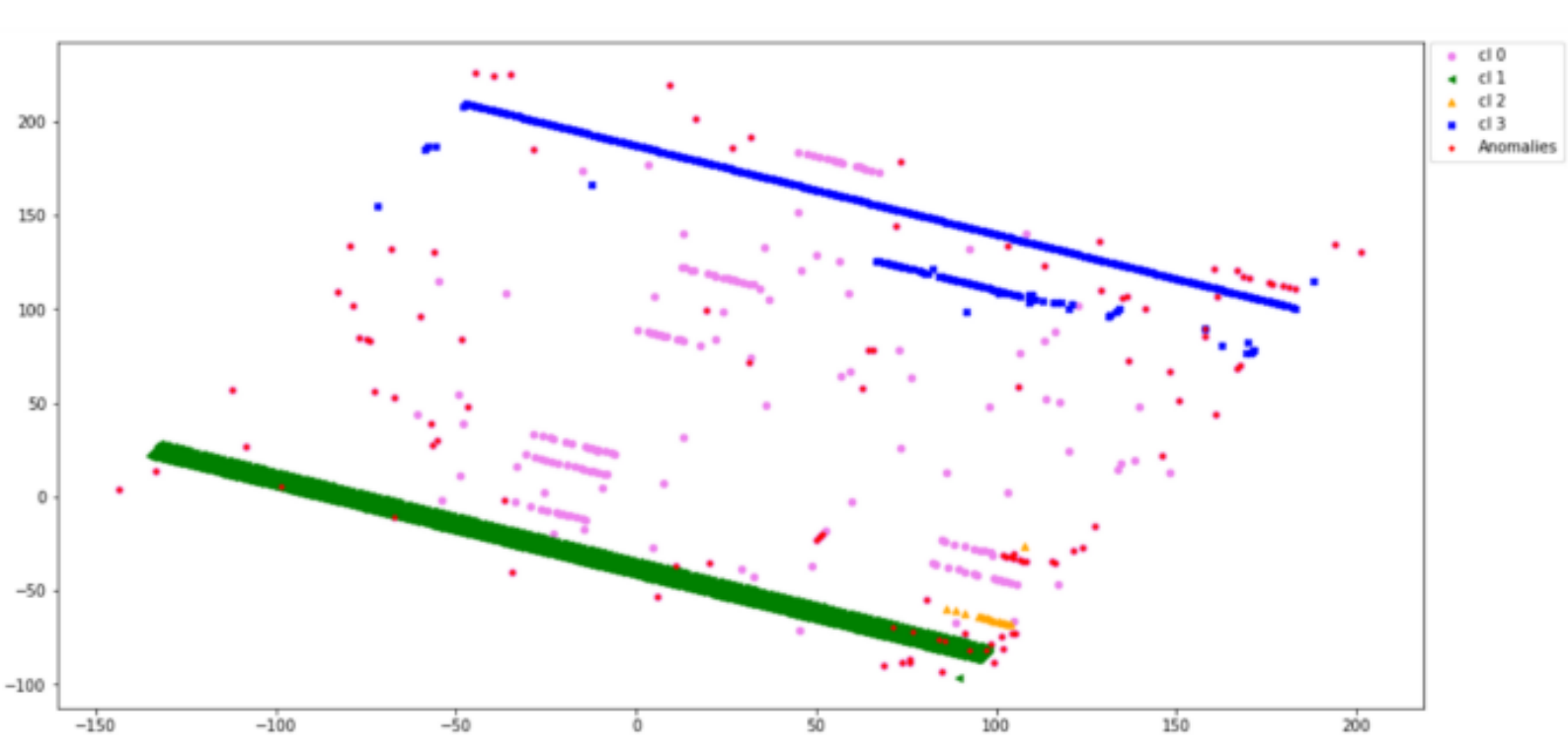
```
107
```

```
In [69]: anomalies_df_d = dst_ip_df[densities_d < dens_threshold_d]
         anomalies_df_d
```

Out[69]:

	index	idst	oct1	oct2	oct3	oct4	pca1	pca2	kcluster
4	4	224.0.0.10	224	0	0	10	84.847	-93.207	0
28	28	172.19.1.100	172	19	1	100	5.893	-53.331	0
30	30	10.7.136.159	10	7	136	159	19.488	99.306	0
33	33	10.61.9.10	10	61	9	10	97.426	-81.867	0
34	34	10.7.137.108	10	7	137	108	65.966	78.434	0

We plot the 107 anomalies detected, along with the clusters as seen below.



We can see that all anomalies are part of the widely scattered cluster 0. We can see their locations plotted above in red.

So, we have successfully clustered both source ip addresses and destination ip addresses using GMM and also detected anomalies in the process.

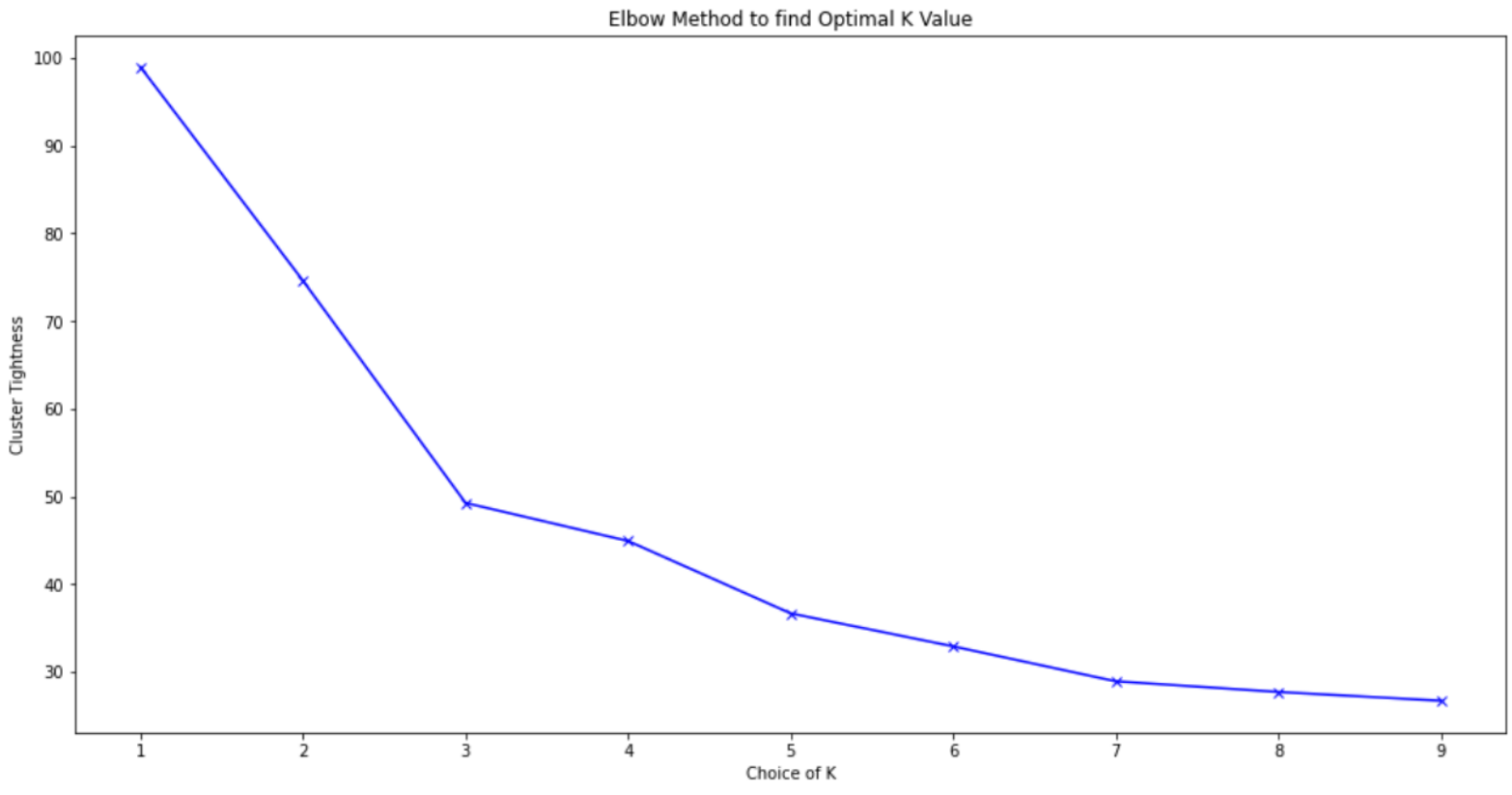
6. Try K-Means Clustering On Destination IP addresses and compare with GMM.

First, we estimate an optimum number of clusters by trying KMeans with different number of clusters and checking the 'elbow' point in the graph between cluster tightness and the number of clusters.

```
In [73]: import numpy as np
         from scipy.spatial.distance import cdist
         from sklearn.cluster import KMeans

         tightness = []
         possibleKs = range(1,10)

         for k in possibleKs:
             km = KMeans(n_clusters=k).fit(X_matrix_dst)
             tightness.append(sum(np.min(cdist(X_matrix_dst, \
                  km.cluster_centers_, 'euclidean'), axis=1)) / X_matrix_dst.shape[0])
```

```
C:\Users\rajar\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:881: UserWa
n Windows with MKL, when there are less chunks than available threads. You can av
_NUM_THREADS=11.
  warnings.warn(
```

```
In [74]: pyplot.plot(possibleKs, tightness, 'bx-')
         pyplot.xlabel('Choice of K')
         pyplot.ylabel('Cluster Tightness')
         pyplot.title('Elbow Method to find Optimal K Value')
         pyplot.show()
```



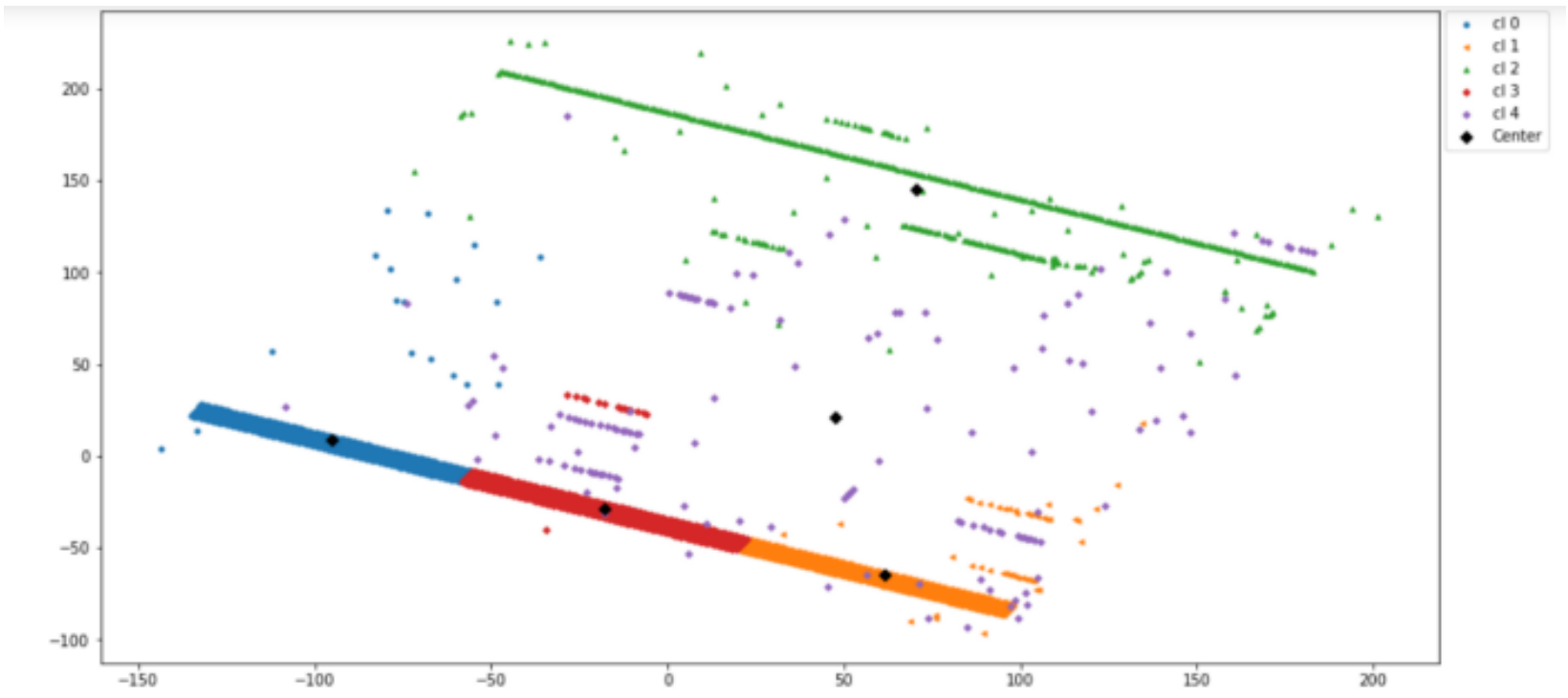
There is no clear one elbow point. We will choose 5 as the optimum number of clusters and go ahead with clustering by K-Means fit_predict on the data points.

```
In [75]: # Fit KMeans with 5 clusters, predict the clusters and save the labels as a new column.
         kclusters=5
         kms = KMeans(n_clusters=kclusters,n_init=200,random_state=0)

         kms.fit_predict(X_matrix_dst)

         dst_ip_df2['kcluster']=kms.labels_.tolist()
```

We plot the clusters identified by K-Means.

Plot the clusters

```
In [79]: plt.rcParams["figure.figsize"] = (16,8)
         pyplot.scatter(dfkms0['pca1'],dfkms0['pca2'],s=10,label="cl 0")
         pyplot.scatter(dfkms1['pca1'],dfkms1['pca2'],s=10,marker="<",label="cl 1")
         pyplot.scatter(dfkms2['pca1'],dfkms2['pca2'],s=10,marker="^",label="cl 2")
         pyplot.scatter(dfkms3['pca1'],dfkms3['pca2'],s=10,marker="D",label="cl 3")
         pyplot.scatter(dfkms4['pca1'],dfkms4['pca2'],s=10,marker="D",label="cl 4")
         pyplot.scatter(centers_df['pca1'], centers_df['pca2'],s=30,marker="D",label="Center", color='Black')

         pyplot.legend(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)
         pyplot.show()
```



We can see that the clustering done by K-Means is different from that by GMM.

1. The ellipsoidal data points at the bottom has been split in to 3 clusters while GMM clustered them as a single big cluster.
2. The clusters don’t seem to be clearly split and contains mixed data points from the scattered data points.

Overall the clustering is not as good and clearly demarcated as done by GMM. This should be because K-Means is based on cluster center method which suits more to spherical blobs of data and not ellipsoidal shapes of different sizes and orientations.

```
In [81]: print(dfkms0.oct1.value_counts())
         print()
         print(dfkms0.oct2.value_counts())
         print()
         print(dfkms0.oct3.value_counts())
```

```
192     682
```

```
192     682
88        2
195       2
208       2
159       1
93        1
87        1
78        1
84        1
172       1
217       1
91        1
72        1
81        1
209       1
77        1
Name: oct1, dtype: int64
```

Looking at the cluster 0 first octet itself, we can see it contains mix of subnets. Though majorly it consists of 192.168, it also contains 208. , 195. etc.

K-Means has broken a homogenous cluster of 192.168.[21–28] subnet IP addresses in to three smaller clusters and also mixed them up with other dissimilar IP addresses from the scattered set.

But if you recall, GMM had finely segregated all 192.168.[21–28] subnet addresses in to one cluster (Cluster 1) and did not mix up other IP addresses. **So clustering by GMM has delivered better results than K-Means for the ellipsoidal shaped groups as expected.**

Conclusion

We used both GMM and K-Means clustering algorithms to cluster IP addresses found in the data set.

We saw that GMM delivered better results particularly for destination IP addresses by clearly demarcating the ellipsoidal data blobs than K-Means.

In the process, we also detected anomalies within the IP addresses used.

Thank You !
Hope this was interesting and helped with a bit of learning !

Reference Materials

- Data Analytics For IT Networks- by John Garret
- Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition

---