Lab2 Report

309552043 楊宇正

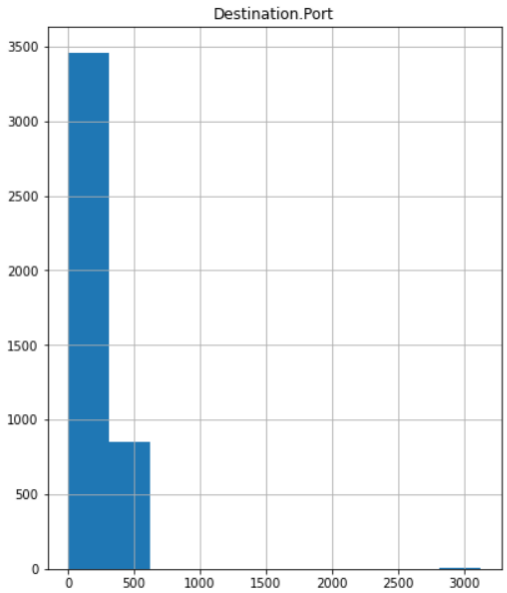
1. Data processing

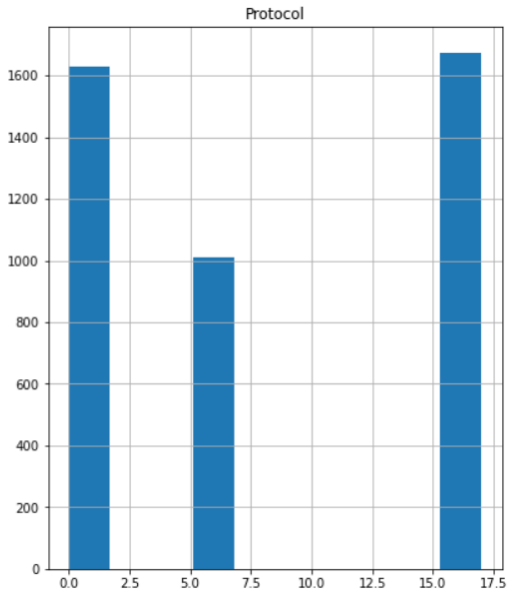
For data preprocessing, I specified some parameters when reading from the raw data CSV file such as **“error_bad_lines=False”**, and the function **“drop_duplicates()”** to filter any invalid, missing, or duplicate data from raw dataset.

Also, I tried to use **K-Fold (K=2)** to split the original dataset to two subsets. However, I didn’t use it in the final code I submitted because I realized that this is an unsupervised learning and there’s no need to split the original dataset to training set and test set. All the data could be used as the training set since we already got the labeled cluster files to evaluate the final score.

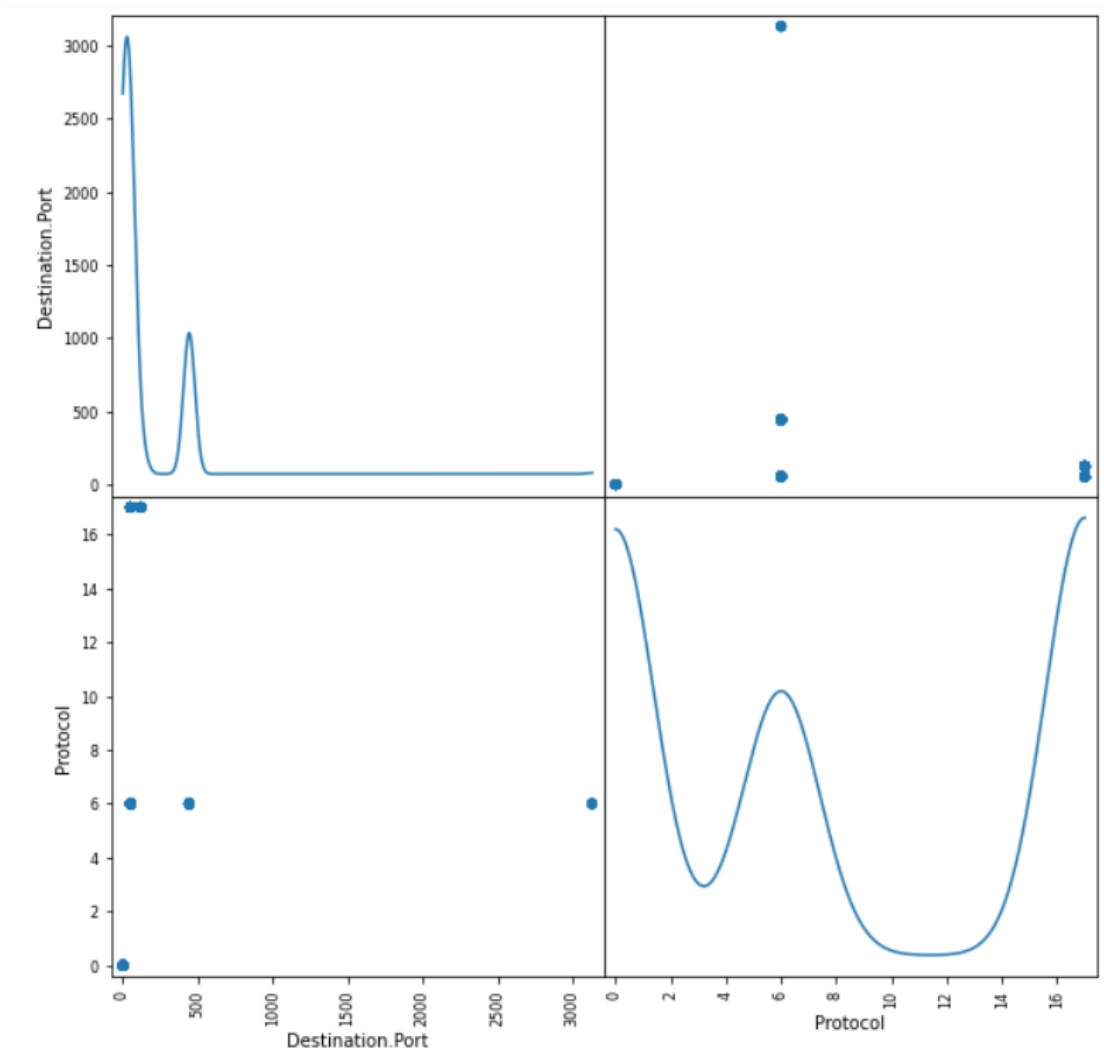
2. Visualize Data

- The distribution of each feature in histogram:





- The scatter matrix of feature “Destination.Port” and “Protocol”:



3. Feature Engineering

For feature selection, I firstly chose only **“Source.Port”**, **“Destination.Port”**, and **“Protocol”**, these three features. Because I believe that these are key features that affect the clustering result the most. (Service type, tcp/udp, etc.) Which is based on my domain knowledge.

Then, I go through the extracted Dataframe and compare the src_port and dst_port for each entry. If the number of the src_port is smaller than dst_port, then swap them. The reason to do so is that as far as I know that the well-known ports are usually smaller than the random source port. To be able to only use dst_port to identify what type of service the flow is, I have to swap the smaller src_port with dst_port since the flow may be a response from server (e.g HTTPS, the src_port may be 443).

```
X = df.filter(items=['Source.Port', 'Destination.Port', 'Protocol'])

## Swap src_port and dst_port if src_port is the smaller (well-known port)
index = -1
for src, dst in zip(X['Source.Port'], X['Destination.Port']):
    index = index +1
    if src < dst:
        # src_port = src, dst_port = dst
        X.iloc[index, 1] = src
        X.iloc[index, 0] = dst
```

After this work was done, I then simply drop the column **"Source.Port"**.
So, the final data frame I used to train the model only has **"Destination.Port"** and **"Protocol"** these two features.

```
      Destination.Port  Protocol
0                  443         6
1                  443         6
2                  443         6
3                  443         6
4                  443         6
...                ...       ...
4312                53        17
4313                53        17
4314                53        17
4315                53        17
4316                53        17

[4317 rows x 2 columns]
Index(['Destination.Port', 'Protocol'], dtype='object')
```

4. Clustering Algorithms & (Together with 5., see below)

5. With different parameters

- **K-Means:**
  - Without Normalization:
    I tried build the model with different parameter **"init" ('k-means++' or 'random').** Both have very similar score in the end.
    The score is around **0.94** which is not bad.

In [9]:
```
# Do K-Means with K-means++ enabled for better initial centroids
kmeans = KMeans(n_clusters=4, init='k-means++', random_state=0)
kmeans.fit(X)
kmeans.labels_
#kmeans.predict(X.iloc[:3])
```

Out[9]:
```
array([1, 1, 1, ..., 3, 3, 3])
```

In [10]:
```
# Evaluation
from sklearn.metrics.cluster import adjusted_mutual_info_score
print(kmeans.labels_.shape)
print(Y.shape)
adjusted_mutual_info_score(Y, kmeans.labels_)
```

```
(4317,)
(4317,)
```

Out[10]:
```
0.9455789198780098
```

In [21]:
```
# KMeans with different parameter (init='random')
kmeans_ran = KMeans(n_clusters=4, init='random', random_state=0)
kmeans_ran.fit(X)
adjusted_mutual_info_score(Y, kmeans_ran.labels_)
```

Out[21]:
```
0.9455789198780099
```

- With Normalization:
  The score is around 0.85, which is lower than the case without normalization.

In [11]:
```
# KMeans with Normalization
## Normalization
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X)
X_nor = scaler.transform(X)
## Build model
kmeans_nor = KMeans(n_clusters=4, init='k-means++', random_state=0)
kmeans_nor.fit(X_nor)
adjusted_mutual_info_score(Y, kmeans_nor.labels_)
```

Out[11]:
```
0.8494223070905361
```

In my experiments, the score with normalization from **"adjusted_mutual_info_score()"** was even lower than without normalization. I think the reason may be that the actual values of Destination.Port and Protocol are not continuous but discrete. So there's may be no need to normalize them actually.

- **Agglomerative Clustering:**

I tried agglomerative clustering with different linkage parameters.

- Linkage=**"Ward"** (Centroid):
  When using "Ward" linkage, the score was pretty close to the score of K-Means:

In [19]:
```
# Agglomerative Clustering (Hierarchical clustering)
from sklearn.cluster import AgglomerativeClustering
agg_linkage = AgglomerativeClustering(n_clusters=4, linkage='ward')
agg_linkage.fit(X)
agg_linkage.labels_

# Evaluation
adjusted_mutual_info_score(Y, agg_linkage.labels_)
```

Out[19]: 0.9455789198780098

- Linkage=**"Single"**
  With "single linkage", the performance wasn't looked as ideal as expected. The score was only around 0.7.

In [24]:
```
# Agglomerative Clustering with linkage="single"
agg_single = AgglomerativeClustering(n_clusters=4, linkage='single')
agg_single.fit(X)
agg_single.labels_
# Evaluation
adjusted_mutual_info_score(Y, agg_single.labels_)
# Plot Dendrogram
#from scipy.cluster.hierarchy import dendrogram
```

Out[24]: 0.6984045110897034

- **DBSCAN**:
  - DBSCAN with default parameters (**eps=0.5, min_samples=5**):

In [28]:
```
# DBSCAN with default parameters (eps=0.5, min_samples=5)
from sklearn.cluster import DBSCAN
dbscan_default = DBSCAN()
dbscan_default.fit(X)
print(dbscan_default.labels_)

# Evaluation
adjusted_mutual_info_score(Y, dbscan_default.labels_)
```

```
[0 0 0 ... 4 4 4]
```

Out[28]: 0.9472274267484223

  - DBSCAN with different parameters (**eps=1,**

min_samples=10):

In [29]:
```
# DBSCAN with different parameters (eps=1, min_samples=10)
dbscan_test = DBSCAN(eps=1, min_samples=10)
dbscan_test.fit(X)
print(dbscan_test.labels_)

# Evaluation
adjusted_mutual_info_score(Y, dbscan_test.labels_)
```

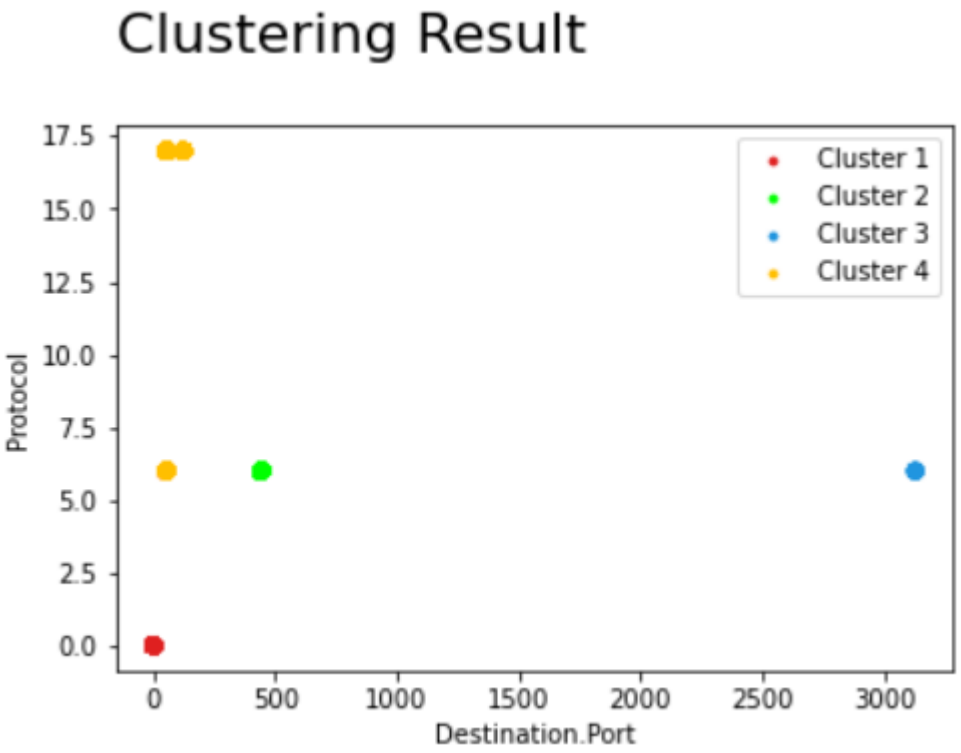
```
[0 0 0 ... 3 3 3]
```

Out[29]:
```
0.9472274267484223
```

- According to above DBSCAN results, seems like the scores are the same for both cases (Both are around **0.94**, not bad). I think the reason could be that the data set I used to fit the model was already obviously centralized (the distribution of Destination.Port and PROTOCOL are very monotonous, e.g., PROTOCOL may only be 6 or 17)

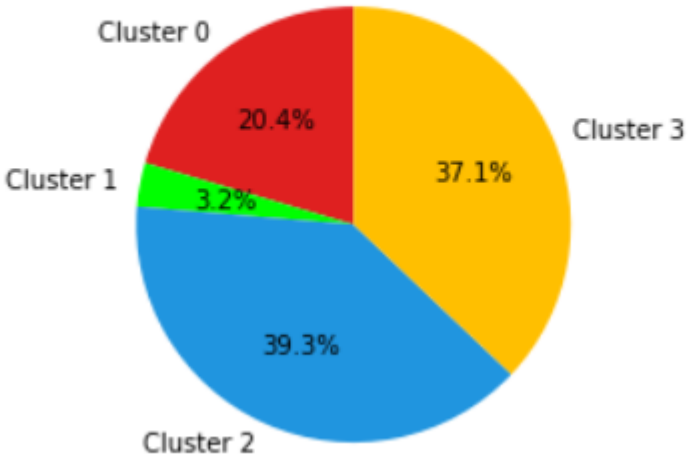
6. Visualize Clusters

- Below is the visualization result with K-Means, since I only picked two features (dst_port & protocol) thus there're many overlapped markers. Also, the distribution of the raw data entries in the original dataset were very monotonous and centralized already. (i.e., lack of variety)



- Also, below is the visualization result of **DBSCAN** presented by a pie chart:
  - The first row of the array in the following figure is the counts of each label predicted by DBSCAN, the second one is the counts of each labels from **"cluster.csv"** provided by TA.
  - As you can see, the result of prediction is quite accurate and the distribution of each cluster is very clear:

```
[849, 135, 1631, 1540]
[856, 135, 1631, 1695]
```



7. Measure Performance

- Please refer to the section 4 & 5, I measured performance and showed the score for each algorithm I used by using the function **“adjusted_mutual_info_score()”** as requested.

8. Cluster the traffic flows with Domain Knowledge

- As mentioned in previous questions, after manually checking the feature list of the original dataset, based on my network domain knowledge, I think that both **“port”** and **“protocol”** may affect clustering result the most, since in most cases, we may be able to know what kind of the service that a flow belongs to by just checking the src_port, dst_port (if it uses the well-known port), and the IP protocol it uses. (e.g.,
  dst_port=443 -> HTTPS,
  dst_port=53, protocol=17 -> DNS)
  And that’s why I only chose src/dst port and protocol as the features and even dropped src_port after checking for well-known ports in the end.

9. Conclusion

- Compare with lab1, lab2 is still interesting for me this time. To be honest, I was not familiar with algorithms for unsupervised learning (clustering) as for classification. It took me some times to search and to know how do they work practically.
- Another interesting observation during this lab was that, the score that I ran K-Means by only selecting **“Destination.Port”** and **“PROTOCOL”** as features directly was very poor. Only around 0.34. The reason behind that could be that some of the flows are server responds (e.g., src_port=**443**, dst_port=**56147**), since the first time I just simply clustered them by dst_port, thus this kind of flows were very likely be clustered to another cluster (however, it should be clustered with those flows using dst_port=**443,** since they’re all HTTPS traffic)
- Overall, this is a fun lab, I did really enjoy it although it took me some times for searching, understanding the algorithms and usage of APIs.