Project Report 2021

MDA COURSE

Mohammadreza Taremi
96101902

Part 1-2 – Preprocess & CF on Cameras

First of all, we need to create a matrix(named *n_traverse* in code) with the shape of (#cameras, 7, 24), to do so I used *np.zeros*. After that, we need to fill in the cells of this matrix with the number of traffic occurred in a particular day of week and hour of day. To extract such data from our dataframe I used the following codes:

```
df_1 = df.withColumn('DAY_OF_WEEK',dayofweek(df.PASS_DAY_TIME))
   df_2 = df_1.withColumn('HOUR', hour(df_1.PASS_DAY_TIME))
```

The output for the first row of dataframe is as shown below:

```
+-----------+-------------+-------------------+-----------+----+
|DEVICE_CODE|FINAL_CAR_KEY|      PASS_DAY_TIME|DAY_OF_WEEK|HOUR|
+-----------+-------------+-------------------+-----------+----+
|        143|     64111706|2021-06-01 01:17:52|          3|   1|
+-----------+-------------+-------------------+-----------+----+
```

Day of week ranges from 1 to 7. (1- Sunday , 2- Monday …… 7- Saturday)

Now we ought to calculate the frequency of each cameras traffic, based off its day_of_week and hour of the day. To do so, I exploited *groupby* method with the key list of *DEVICE_CODE, DAY_OF_WEEK, and HOUR* as follows:

```
df_3 = df_2.groupby(['DEVICE_CODE','DAY_OF_WEEK','HOUR']).count()
```

The result (df_3) is shown below:

```
+-----------+-----------+----+-----+
|DEVICE_CODE|DAY_OF_WEEK|HOUR|count|
+-----------+-----------+----+-----+
|     900182|          3|   6|    1|
|     900159|          3|   8|    1|
|     631833|          3|   4|    1|
```

#traffic observed in the 6th hour of the 3rd day of the week by camera 900182 = 1.

In the next part, we have to fill in the *n_traverse matrix* with the values within *df_3* dataframe. To do that, I performed the following steps:

1. Creating a list which include all the unique camera codes

```
cameras = [x.DEVICE_CODE for x in df_2.select('DEVICE_CODE').distinct().collect()]
```

2. Creating a dictionary whose keys and values are camera codes and their corresponding index respectively.

```
cam_dict = {}
for i, camcode in enumerate(cameras):
cam_dict[i] = camcode
cam_dict = {v: k for k, v in cam_dict.items()}
cam_dict
>>>
{100: 181,
 113: 101,
 114: 85,
 115: 14,
 117: 172,
 118: 133,
 119: 128,
 135: 124,
```

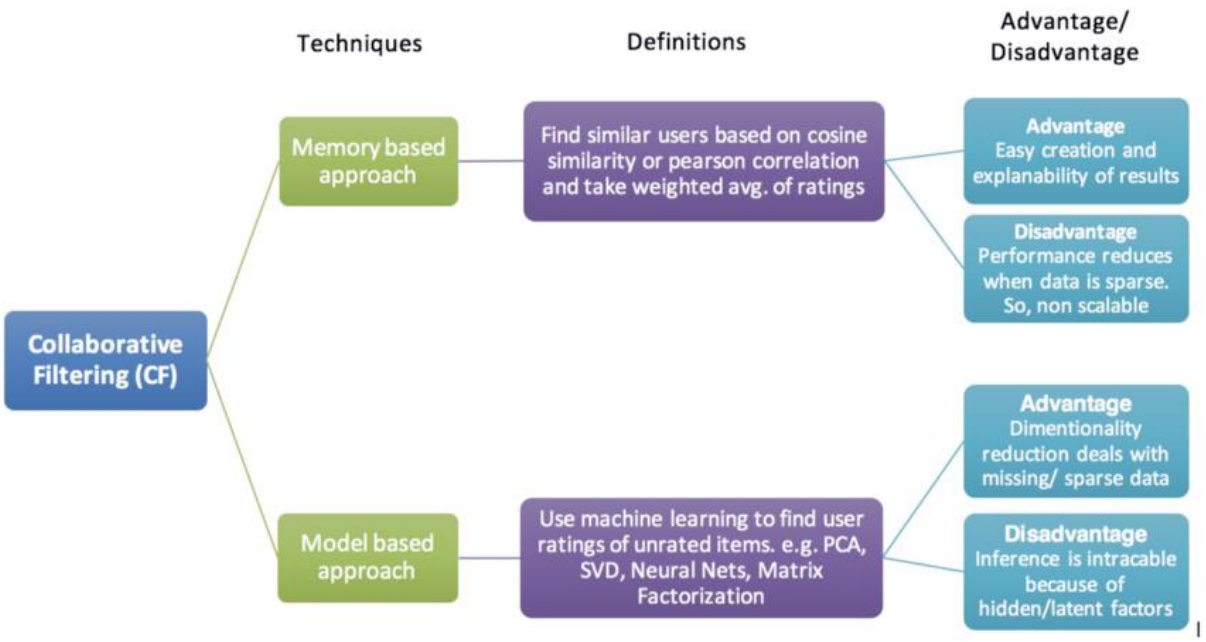
3. Iterating over each of elements of *df_*3 dataframe with the camera codes, day of weeks, hour and counts and fill the corresponding cell of the 3D *n_traverse* matrix.

```
for cam,d,h,count in df_3.collect():
n_traverse[cam_dict[cam],d-1,h-1] = count
```

> *n_traverse* matrix description:
> 1st dimension is assigned for a specific camera
> 2nd dimension is assigned for days of weeks
> 3rd dimension is assigned for hour of the day

Now that we have computed each of the camera's corresponding traffics, we can implement Collaborative Filtering Method to find the similar cameras. As a matter of fact, cameras in this context are playing the role of users in famous Movie Recommendation System problem, and also the number of traffics in each hour and day of the week in this problem are indeed the ratings.

As we know, CF techniques are broadly divided into 2-types:



The first category includes algorithms that are memory based, in which statistical techniques are applied to the entire dataset to find the similar users. Memory-Based Collaborative Filtering approaches can be divided into two main sections: **user-item** filtering and **item-item** filtering. As we know, a **user-item filtering** takes a particular user, find users that are similar to that user based on similarity of ratings. Therefore, we need this type of CF in Memory Based Approach. For similarity metric I used **Pearson Correlation Coefficient**. To find the cameras with higher correlation coefficient, firstly, we have to create a matrix with the shape of (#cameras, 24×7). Each of this matrix's row is the concatenated version of *n_traverse* matrix. The following code does the trick for us:

```
user_item_mat = np.zeros(shape=(len(cameras),7*24))
for i in range(user_item_mat.shape[0]):
  user_item_mat[i] = n_traverse[i].ravel()

user_item_df = pd.DataFrame(user_item_mat)
corr_mat = user_item_df.T.corr()
corr_mat
>>>
```

	0	1	2	3	4	5	6	7	8	9	
0	1.000000	0.222621	0.276549	0.441643	0.123884	0.364643	0.169325	0.104620	0.267636	0.364265	0.
1	0.222621	1.000000	0.280842	0.221848	0.407088	0.035528	0.221846	0.443416	0.288646	0.291300	0.
2	0.276549	0.280842	1.000000	0.784631	0.677693	0.845443	0.689903	0.535816	0.851348	0.910645	0.
3	0.441643	0.221848	0.784631	1.000000	0.485929	0.756188	0.727829	0.492317	0.831720	0.886924	0.
4	0.123884	0.407088	0.677693	0.485929	1.000000	0.599717	0.530667	0.738761	0.492796	0.677469	0.
...	...	...	...	...	...	...	...	...	...	...	
712	0.171289	0.361072	0.820998	0.727377	0.619242	0.742171	0.649534	0.540166	0.675726	0.836277	0.
713	0.093385	0.279779	0.763817	0.698786	0.688732	0.746100	0.758011	0.589879	0.616647	0.833204	0.
714	-0.010434	-0.009828	0.123241	0.465534	-0.017379	0.180398	0.157973	-0.016374	0.376420	0.240600	0.

Now that we have calculated the correlation matrix, we can extract the higher correlation coefficients, and similar cameras afterwards. The index of the values in the matrix above determines the corresponding camera index, and by having that index and using the dictionary we defined earlier we can extract the DEVICE_CODE of similar cameras.

1. First we extract the upper triangle of correlation matrix with the following code:

```
corr_df = np.tril(corr_df,-1)
```

	0	1	2	3	4	5	6	7	8
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.222621	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2	0.276549	0.280842	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
3	0.441643	0.221848	0.784631	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
4	0.123884	0.407088	0.677693	0.485929	0.000000	0.000000	0.000000	0.000000	0.000000
...	...	...	...	...	...	...	...	...	...
712	0.171289	0.361072	0.820998	0.727377	0.619242	0.742171	0.649534	0.540166	0.675726
713	0.093385	0.279779	0.763817	0.698786	0.688732	0.746100	0.758011	0.589879	0.616647
714	-0.010434	-0.009828	0.123241	0.465534	-0.017379	0.180398	0.157973	-0.016374	0.376420
715	0.170089	0.043387	0.468517	0.757840	0.313338	0.490114	0.491133	0.227994	0.648088

2. Second, we get the indices of top 100 values in the correlation matrix:

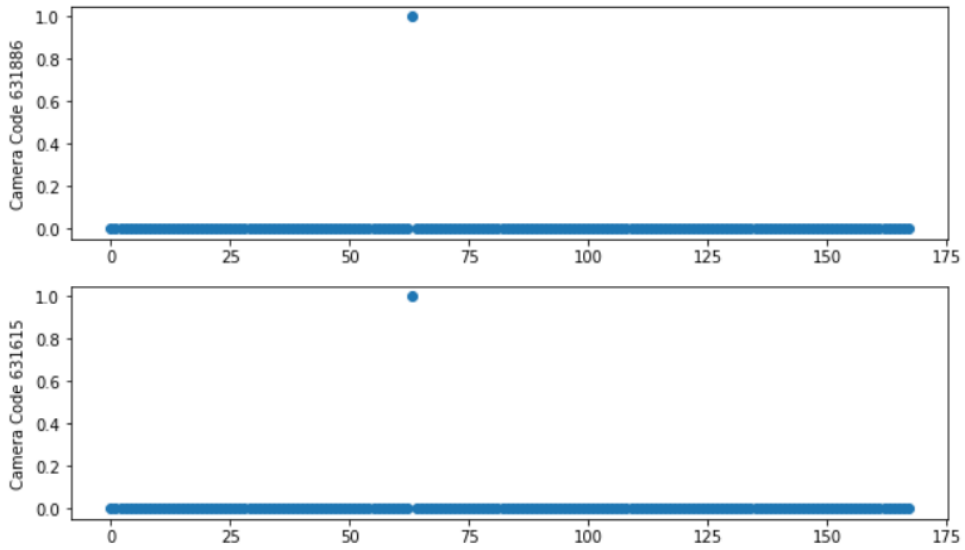
```
idx_0 = idx[0].tolist()
idx_1 = idx[1].tolist()
camera_series = pd.Series(cameras)
accessed_series_0 = camera_series[idx_0].values
accessed_series_1 = camera_series[idx_1].values
sim_cameras = pd.DataFrame({'Camera 1': accessed_series_0, 'Camera 2': list
(accessed_series_1)}, columns=['Camera 1', 'Camera 2'])
```

```
sim_cameras
>>>
```

	Camera 1	Camera 2
0	631886	631615
1	631117	202001
2	635526	174
3	631937	100700920
4	631937	100700979
...	...	...
95	631997	1001024
96	631997	631881
97	100700971	635608
98	631886	605511

This table indicates that Camera (with code 631886) is similar to Camera (with code 631615)

To compare the result, I scattered each of these cameras corresponding $\mathbf{24} \times \mathbf{7}$ vector:



As can be seen, these cameras observes same number of traffics in the entire week.

Part 3 – CF for Cars

In this part, firstly, we have to find all the unique paths. Then, for every unique car, we find a sparse vector which contains 1s & 0s. 1 iff the path exists in the *paths* vector, and 0 if not.
By doing this, we can calculate the similarity among the cars.

1. To find the paths which belong to one car, we have to use *groupBy* and *agg* method. First we set the FINAL_CAR_KEY as key and its corresponding DEVICE_CODE as values. By doing so, we can create the following dataframe:

```
df_4 = df_2.groupBy('FINAL_CAR_KEY').agg(F.collect_list("DEVICE_CODE"))
df_4 >>>
```

FINAL_CAR_KEY	collect_list(DEVICE_CODE)
100001221	[631633]
100002495	[200301]
10000313	[900276]
10000529	[22010122]
10000584	[900271]
10001007	[100700866, 900155]
10001148	[22010061]
10001203	[100700845]
10001483	[900241]
100015798	[212802]

Based on the table above, the car with FINAL_CAR_KEY of 10001007 has been observed by two cameras with codes, 100700866 and 900155

2. Then for each camera code of a specific car, we set the corresponding value of *adj_mat* to 1 and 0 otherwise. For example for the car 10001007 we have:

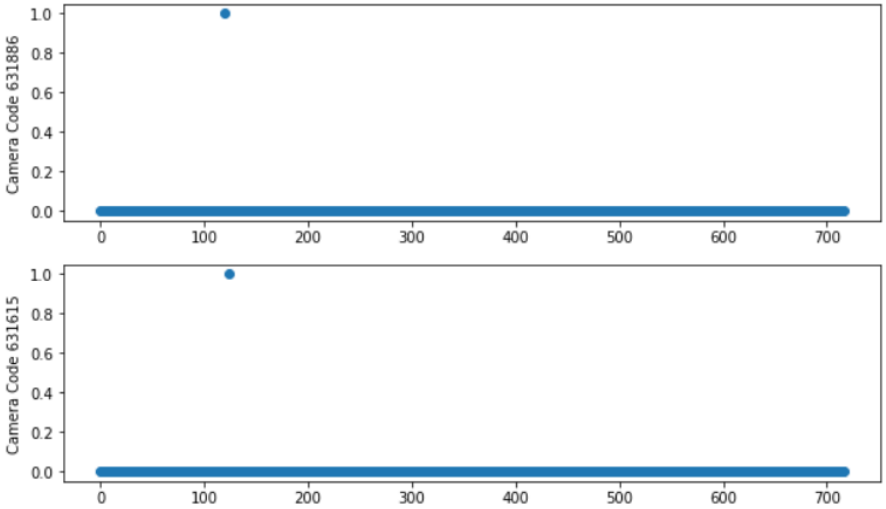
```
adj_mat[car_dict['10001007']][paths.index(900155)]
>>>
(27333, 532)
So in adj_mat[27333,532] we must have the value of 1. To check that:
adj_mat[car_dict['10001007']][paths.index(900155)]
>>>
1
Which is the true value.
```

To find the most travelled cars, we need to select the rows of *adj_mat* which have the sum of column values greater than a *threshold*. This means that if a corresponding camera vector of a car has more than *threshold* 1s, then, it's considered as a top travelled car. Like the previous part, we calculate the correlation and select the most similar cars based off their correlation coefficient. The code snippet below, does the trick for us:

```
Threshold = 2
car_path_df = car_path_df[car_path_df.iloc[:, 1:].sum(axis=1)>=threshold]
car_path_df = pd.DataFrame(adj_mat)
corr_df_2 = car_path_df.T.corr()
corr_df_2 = np.tril(corr_df_2,-1)
idx_2 = np.unravel_index(np.argsort(corr_df_2.ravel())[-
100:], corr_df_2.shape)
idx_0_2 = idx_2[0].tolist()
idx_1_2 = idx_2[1].tolist()
camera_series = pd.Series(cars)
accessed_series_0_2 = camera_series[idx_0_2].values
accessed_series_1_2 = camera_series[idx_1_2].values
sim_cars = pd.DataFrame({'Car 1': accessed_series_0_2, 'Car 2': list(accessed_
series_1_2)},
                        columns=['Car 1', 'Car 2'])
sim_cars >>>
```

	Car 1	Car 2
0	16225843	9254994
1	21476546	11226740
2	7824990	9350714
3	28585770	88160685
4	11687564	90734667
...	...	...
95	7680591	17655565
96	9454816	15410481

For the first pair of cars we have the following plots:

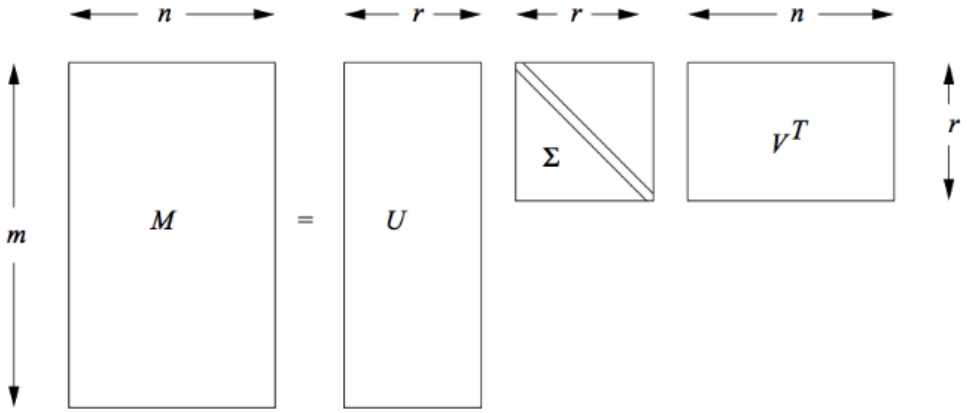


Conclusion:

As we have defined, *adj_mat* is a sparse matrix. Therefore working with such matrix for various purposes such as computing correlation coefficient takes up a lot of time. One disadvantage of Memory Based Collaborative Filtering approach that we utilized in this section and the section before is the amount of time they required. Therefore we need to use other methods such as Model Based Collaborative Filtering to find similar cameras, or cars (users) to tackle this problem.

Part 4 – Utility Matrix & SVD

We'll use the Singular Value Decomposition, or SVD for short, in this section. According to the project guidelines, we must create a utility matrix based on the most frequently traveled cars and cameras. In this section, I thought of a matrix in which the rows represent the most traveled cars and the columns represent the encoded camera codes, with 1 if the corresponding car has been observed by that camera and 0 if it has not.



The utility matrix (before normalization) is as follows:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
45	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
49	0.0	0.0	0.0	0.0	0.0	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	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
66	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
43364	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
43373	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	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	0.0	0.0	0.0	0.0	0.0	0.0	0.0
43377	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
43448	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
43449	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

It’s obvious that this matrix is a sparse one, and since we don’t need all of its columns we can perform dimensionality reduction approaches such as SVD. The implementation of SVD includes Covariance Matrix calculation and then computing the eigenvectors and values:

```
normalised_mat = car_path_df - np.asarray([(np.mean(car_path_df, 1))]).T

#utility matrix
A = normalised_mat.T / np.sqrt(car_path_df.shape[0] - 1)
```

```
from scipy.linalg import svd as scipy_svd
U, S, V = scipy_svd(A)
pd.DataFrame(np.diag(S))
>>>
```

	0	1	2	3	4	5	6
0	0.389204	0.000000	0.000000	0.000000	0.000000	0.0	0.0
1	0.000000	0.212364	0.000000	0.000000	0.000000	0.0	0.0
2	0.000000	0.000000	0.188161	0.000000	0.000000	0.0	0.0
3	0.000000	0.000000	0.000000	0.178595	0.000000	0.0	0.0
4	0.000000	0.000000	0.000000	0.000000	0.178459	0.0	0.0
...	...	...	...	...	...	...	...

As can be seen, we can set a threshold for the diagonal values of Singular matrix and by doing so we can reduce the number of dimensions.

It's worth mentioning that power iteration algorithm was implemented but due to its long run time even for a sample of dataset, I decided to use built-in functions:

```
def power_svd(A, iters):
      mu, sigma = 0, 1
      x = np.random.normal(mu, sigma, size=A.shape[1])
      B = A.T.dot(A)
      for i in range(iters):
          new_x = B.dot(x)
          x = new_x
      v = x / np.linalg.norm(x)
      sigma = np.linalg.norm(A.dot(v))
      u = A.dot(v) / sigma
      return np.reshape(
          u, (A.shape[0], 1)), sigma, np.reshape(
          v, (A.shape[1], 1))
def main():
      t = time.time()
      A = np.array([[1, 2], [3, 4]])
      rank = np.linalg.matrix_rank(A)
      U = np.zeros((A.shape[0], 1))
      S = []
      V = np.zeros((A.shape[1], 1))

      # Define the number of iterations
      delta = 0.001
      epsilon = 0.97
      lamda = 2
      iterations = int(math.log(
          4 * math.log(2 * A.shape[1] / delta) / (epsilon * delta)) / (2 * lamda))
```

```
# SVD using Power Method
for i in range(rank):
    u, sigma, v = power_svd(A, iterations)
    U = np.hstack((U, u))
    S.append(sigma)
    V = np.hstack((V, v))
    A = A - u.dot(v.T).dot(sigma)
```

Part 5 – Clustering

The first step we need to take in every clustering problem is to define feature matrix. For feature matrix we select the *n_traverse* matrix which is composed of 7*24 vectors for each camera and each element in this vectors indicates the number of traffics in a specific hour of the particular day. The code snippet below is written to cluster the cameras based on their 7*24 vector:

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler

feature_mat = user_item_mat
scaler = StandardScaler()
scaled_features = scaler.fit_transform(feature_mat)

kmeans_kwargs = {
    "init": "random",
    "n_init": 10,
    "max_iter": 300,
    "random_state": 42,
}

# A list holds the SSE values for each k
sse = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, **kmeans_kwargs)
    kmeans.fit(scaled_features)
    sse.append(kmeans.inertia_)
```

Based on this image, 10 label is a good choice for the number of centroids. The predicted labels for all the cameras are:

```
kmeans.labels_
>>>
          array([8, 8, 8, 8, 8, 8, 2, 1, 8, 8, 8, 8, 7, 3, 8, 4, 0, 9, 8, 8, 8, 8,
                 1, 3, 3, 8, 8, 8, 9, 8, 8, 8, 8, 8, 1, 1, 8, 8, 8, 8, 8, 9, 9, 3,
                 8, 8, 5, 3, 1, 4, 2, 2, 8, 0, 8, 8, 8, 8, 8, 3, 8, 8, 9, 8, 2, 2,
                 8, 8, 3, 8, 8, 8, 5, 5, 8, 3, 8, 8, 1, 8, 8, 8, 8, 8, 3, 8, 8, 8,
                 3, 7, 3, 8, 8, 8, 8, 8, 8, 8, 2, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 3,
                 8, 8, 8, 8, 2, 8, 8, 8, 8, 2, 3, 8, 8, 8, 0, 2, 8, 8, 8, 4, 2, 8,
                 8, 8, 3, 8, 9, 8, 8, 2, 2, 9, 8, 8, 8, 2, 8, 8, 8, 3, 8, 8, 3, 8,
                 8, 8, 8, 8, 8, 8, 1, 3, 8, 2, 8, 8, 5, 8, 8, 8, 8, 8, 8, 1, 8, 8,
                 8, 8, 8, 9, 8, 8, 8, 8, 8, 1, 8, 8, 8, 8, 8, 9, 8, 8, 0, 8, 4, 2,
                 8, 8, 8, 8, 2, 8, 9, 8, 8, 8, 8, 8, 8, 8, 8, 3, 8, 3, 3, 8, 8, 8,
                 8, 2, 9, 8, 8, 8, 8, 6, 5, 2, 4, 1, 8, 8, 8, 8, 3, 8, 8, 3, 8, 8,
                 8, 8, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 0, 8, 8, 8, 8, 8, 8, 1, 8, 3,
                 8, 8, 8, 8, 8, 8, 3, 2, 8, 8, 5, 9, 5, 3, 8, 8, 2, 8, 8, 8, 8, 1,
                 4, 4, 2, 8, 8, 1, 1, 2, 8, 8, 3, 1, 8, 1, 8, 8, 8, 8, 8, 8, 8, 8,
                 8, 8, 3, 8, 9, 8, 8, 8, 8, 3, 3, 8, 8, 8, 2, 3, 8, 8, 8, 8, 8, 6,
                 8, 2, 9, 8, 8, 8, 2, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 6, 2, 8,
                 8, 8, 3, 0, 8, 8, 8, 8, 8, 8, 2, 8, 8, 8, 8, 3, 8, 9, 4, 4, 8, 8,
                 1, 8, 8, 3, 8, 8, 8, 8, 2, 8, 8, 1, 8, 9, 8, 8, 8, 8, 3, 9, 8, 1,
                 8, 8, 8, 3, 8, 8, 8, 8, 8, 2, 3, 0, 8, 8, 8, 8, 8, 8, 8, 8, 8, 3
```

Now if we split the cameras based off their predicted class label, we may get a better intuition about the feature space of cameras.

For example, the following column includes all of the cameras which are labeled as 1:

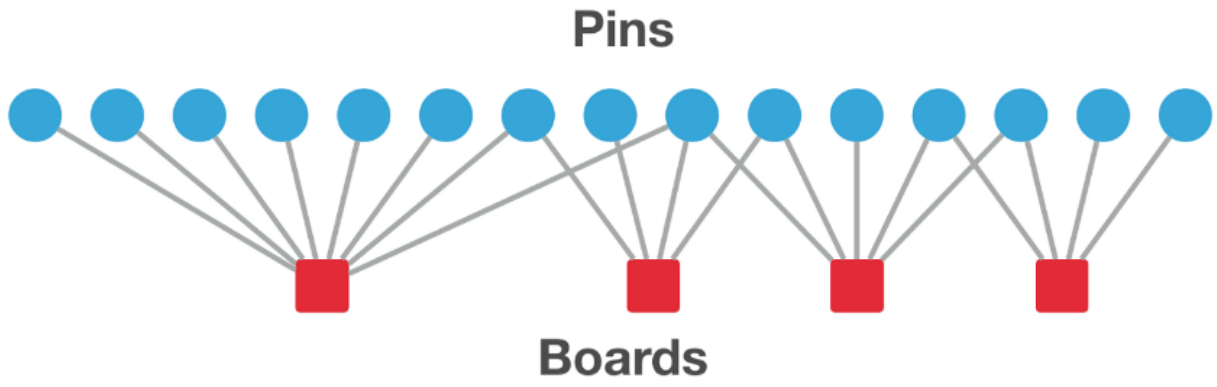
```
pd.DataFrame(label1, columns=['1'])
>>>
```

1
900203
631363
900124
900273
900108
100700826
900185
100701119
631829
900217
22010047
100700864
100700862
900249

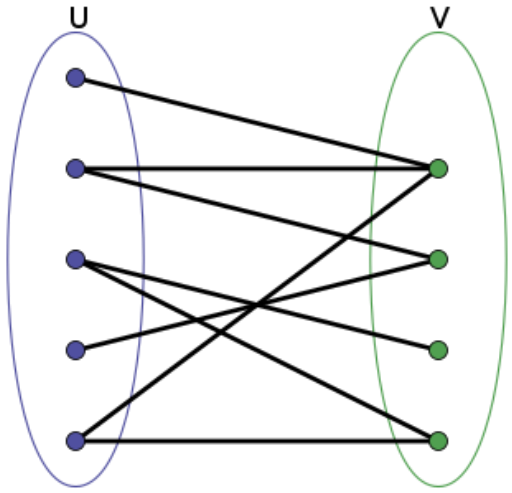
This column shows that cameras 900203, 631363 and …. are in the same group. If we knew how this codes are assigned to each cameras, then it might be possible to get further intuition of clusters and centroids.

Part 6 – Pixie

As we know, in Pixie method which is a graph-based recommendation system we can use a bipartite graph as follows:



Where each edge shows that a person saved a Pin to a board. We can inspire from this idea and develop an approach. At first step, we introduce a bipartite graph:



Where each edge indicates that a car has been observed by a specific camera. Now we have to consider the next part which is Random Walk. Let's consider a single starting pin q. We can simulate a number of random walks starting from q, and record the visit count for each candidate pin the algorithm visits. The higher the visit count for a pin, the more related it is to the original seed pin q.

```
BasicRandomWalk(q: Query pin, E: Set of edges, α: Real, N: Int)
1:  totSteps = 0, V = 0⃗
2:  repeat
3:      currPin = q
4:      currSteps = SampleWalkLength(α)
5:      for i = [1 : currSteps] do
6:          currBoard = E(currPin)[rand()]
7:          currPin = E(currBoard)[randNeighbor()]
8:          V[currPin]++
9:      totSteps += currSteps
10: until totSteps ≥ N
11: return V
```

Now we now need to apply four enhancements to the basic procedure above:

- Biasing the walk towards user-specific pins
- Using multiple weighted query pins
- Boosting pins relating to multiple query pins
- Early stopping to minimize the number of steps (time) while maintaining result quality

Each pin in the query set is assigned a different weight based on the time and type of interactions the user has had with the pin. The random walk algorithm is run for each query pin in the set, with each run capturing its own set of visit counts.

For each pin a scaling factor s_q is calculated as follows, where C is the maximum pin degree in the graph.

$$s_q = |E(q)|.(C - \log|E(q)|)$$

Using the scaling factor, the number of steps N_q allocated to a query starting from pin q with weight w_q is given by:

$$N_q = w_q N \frac{s_q}{\sum_{r \in Q} s_r}$$

Candidates with high visit counts from multiple query pins are considered more relevant to the query than candidates that may have an equally high visit count but where all of those visits come from a single query pin. Thus rather than simply aggregating visit counts across all seed pins, the following formula is used to combine them:

$$V[p] = \left(\sum_{q \in Q} \sqrt{V_q[p]} \right)^2$$

Part 7 – Hubs/Authorities

To apply this method, we have to do the following steps:

1. Initialize the hub and authority of each node with a value of 1
2. For each iteration, update the hub and authority of every node in the graph
3. The new authority is the sum of the hub of its parents
4. The new hub is the sum of the authority of its children
5. Normalize the new authority and hub

Therefore we initialize the hub and authority in the Node constructor. The initial value of auth and hub is 1:

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.parents = []
        self.auth = 1.0
        self.hub = 1.0
```

Now we have to do the following steps:

- For every node in the graph
- Update authority
- Update hub
- Normalize the updated value

```
def HITS_one_iter(graph):
    node_list = graph.nodes

    for node in node_list:
        node.update_auth()
    for node in node_list:
        node.update_hub()

    graph.normalize_auth_hub()
```

Now it's time to compute the new hub and auth values:

- New auth = the sum of the hub of all of its parents
- New hub = the sum of the auth of all of its children

We do this by using the following code snippet:

```
def update_auth(self):
    self.auth = sum(node.hub for node in self.parents)

def update_hub(self):
    self.hub = sum(node.auth for node in self.children)
```

After computing the new values of hubs and auth, it's time to normalize them:

```
def normalize_auth_hub(self):
    auth_sum = sum(node.auth for node in self.nodes)
    hub_sum = sum(node.hub for node in self.nodes)

    for node in self.nodes:
        node.auth /= auth_sum
        node.hub /= hub_sum
```

*** Because of the high computation processes, my virtual RAM in google colab has been used up and couldn't get the final results. I couldn't either complete the last two part of project due to corona virus infection***