BDA 502

TERM PROJECT REPORT

CUSTOMER SEGMENTATION

FERAY ECE TOPCU

MEF University

ISTANBUL

2018

Table of Contents

1. INTRODUCTION

The company I worked is a retail company based on electronic device sector. I am working as a system analyst on a sales application which is used for sales workflow from generating order to printing bill by sales consultants and it is used in all stores around Turkey. Therefore, this sales software generates huge data from these orders which includes order, product and customer information.

I have decided to focus on customer dataset for my term project of machine learning class because after new law of confidentiality of personal data, old segmentation will be corrupted. The company needs new segmentation based on rest of customers who allows the company to keep their data. Because of new obligations caused by the law, I should make all customer data anonymous.

I plan to do behavioral segmentation. Therefore, dataset includes recency, frequency and monetary information for each customer that allows to analyse their data after GDPR. Additionally, I add their favorite product category and brand.

DATASET

Dataset includes recency, frequency and monetary (RFM) information of customer. Additionally, favorite segment and brand of customer columns are available on this dataset. It includes 13897 customers with following details.

Column details can be examined on the table as below:

Column Name	Description	Data Type
CUST_ID	Manipulated Customer ID	Integer
RECENCY	Number of day after last order of customer	Integer
FREQUENCY	Number of total orders of customer	Integer
MONETARY	Total amounts of all orders of customer	Integer
SEGMENT	Favorite category of customer	String
BRAND	Favorite brand of customer	String

2. EXPLORATORY DATA ANALYSIS

- Read CSV file into a dataframe to analyze it.

```
filename = 'data_full.csv'
data = pd.read_csv(filename)
```

- Check column names:

```
data.columns
```

```
Index(['CUST_ID', 'RECENCY', 'FREQUENCY', 'MONETARY', 'SEGMENT', 'BRAND'],
dtype='object')
```

- Data shape:

```
data.shape
(13897, 6)
```

- Check data:

```
data.head()
```

	CUST_ID	RECENCY	FREQUENCY	MONETARY	SEGMENT	BRAND
0	1	2	4	2032	TELEFON	ASUS
1	2	122	7	6226	PC SARF MALZEME	HP
2	3	32	11	4434	PC	SAMSUNG
3	4	94	2	2098	TELEFON	SONY
4	5	41	3	1393	AKSESUAR	PREO

- Firstly, **data.info()** is used to check datatype of each column. As it is seen below, just SEGMENT and BRAND column is float and other columns are categorical.

```
data.info()
```

```
RangeIndex: 13897 entries, 0 to 13896
Data columns (total 6 columns):
CUST_ID      13897 non-null int64
RECENCY      13897 non-null int64
FREQUENCY    13897 non-null int64
MONETARY     13897 non-null int64
SEGMENT      13897 non-null object
BRAND        13892 non-null object
dtypes: int64(4), object(2)
memory usage: 651.5+ KB
```

- When **data.describe()** command is executed, the results are as below:

```
print(data.describe())
```

	CUST_ID	RECENCY	FREQUENCY	MONETARY
count	13897.00000	13897.000000	13897.000000	1.389700e+04
mean	6949.00000	67.426495	5.571994	4.968235e+03
std	4011.86268	40.438236	7.888582	2.747491e+04
min	1.00000	2.000000	1.000000	0.000000e+00
25%	3475.00000	31.000000	2.000000	1.334000e+03
50%	6949.00000	68.000000	4.000000	3.262000e+03
75%	10423.00000	103.000000	7.000000	6.108000e+03
max	13897.00000	138.000000	223.000000	3.084125e+06

- For describing categorical columns below code chunk is executed:

```
categorical = data.dtypes[data.dtypes == "object"].index
print(data[categorical].describe())
```

	SEGMENT	BRAND
count	13897	13897
unique	40	257
top	TELEFON	SAMSUNG
freq	5261	3509

- Nullity check for each column:

```
#nullity check:
print("Nullity Check:\n",pd.isnull(data).any())
print("Sum of null values for each column:\n",pd.isnull(data).sum())
```

```
Nullity Check:
 CUST_ID      False
RECENCY       False
FREQUENCY     False
MONETARY      False
SEGMENT       False
BRAND         False
```

There is no null value on any column.

- The result of describing categorical columns show that there are different segments in dataset. Check what are they with the following code chunk:

```
#Find unique SEGMENTS:
categorical = data.dtypes[data.dtypes == "object"].index
print(categorical)
print(data[categorical]["SEGMENT"].describe())
unique_segment = data.groupby('SEGMENT')["CUST_ID"].nunique()
print("Unique Brands:\n",unique_segment)
```

AKILLI CIHAZLAR	12
AKSESUAR	803
AYDINLATMA CIHAZI	10
CEP TELEFONU AKSESUAR	467
CEVRE BIRIMLER	857
DİJİTAL GÖRÜNTÜLEME AKSESUAR	9
DİJİTAL GÖRÜNTÜLEME CIHAZ	38
ELEKTRIKLI SUPURGE	120
EV SINEMA SISTEMLERI	26
GORUNTU AKSESUAR	49
GORUNTU SISTEMLERI	991
GİYİLEBİLİR AKSESUAR	37
HIZMET PAKETI	490
ISITICI	12
KABLO AKSESUAR	120
KISISEL BAKIM	675
KLİMA	2
KONDISYON VE SAGLIK URUNLERI	1
KONSOL	61
KONSOL AKSESUAR	59
KONSOL OYUN	85
KÜÇÜK EV ALETLERİ	388
NAVIGASYON CIHAZLARI	10
OEM	1
OUTDOOR	1
OYUNCAK	89
OYUNCAKLAR	2
PC	1383
PC KOMPONENT	232
PC SARF MALZEME	377
PC YAZILIM	17
PIL GRUBU	327
SES SISTEMLERI	105
SOLO	138
SOLO-MDA YENİ İŞ MODELİ	2
TABLET AKSESUAR	13
TELEFON	5261
VANTİLATÖR	10
YAZICI	296

It is required to clean these types as shown above:

KONSOL and KONSOL OYUN

PC and PC KOMPONENT

SOLO and SOLO-MDA Yeni İş Modeli

are same category, they should be cleaned.

- Clean the SEGMENT column:

```
#Cleaning segment data:

sb_index=np.where(data["SEGMENT"]=='KONSOL OYUN')
for i in sb_index:
    data.loc[i,"SEGMENT"]='KONSOL'

sb_index=np.where(data["SEGMENT"]=='PC KOMPONENT')
for i in sb_index:
    data.loc[i,"SEGMENT"]='PC'

sb_index=np.where(data["SEGMENT"]=='SOLO-MDA Yeni İş Modeli')
for i in sb_index:
    data.loc[i,"SEGMENT"]='SOLO'


print(data[categorical].describe())
```

	SEGMENT	BRAND
count	13897	13897
unique	38	257
top	TELEFON	SAMSUNG
freq	5261	3509

- The result of describing categorical columns show that there are different brands in dataset. Check what are they with the following code chunk:

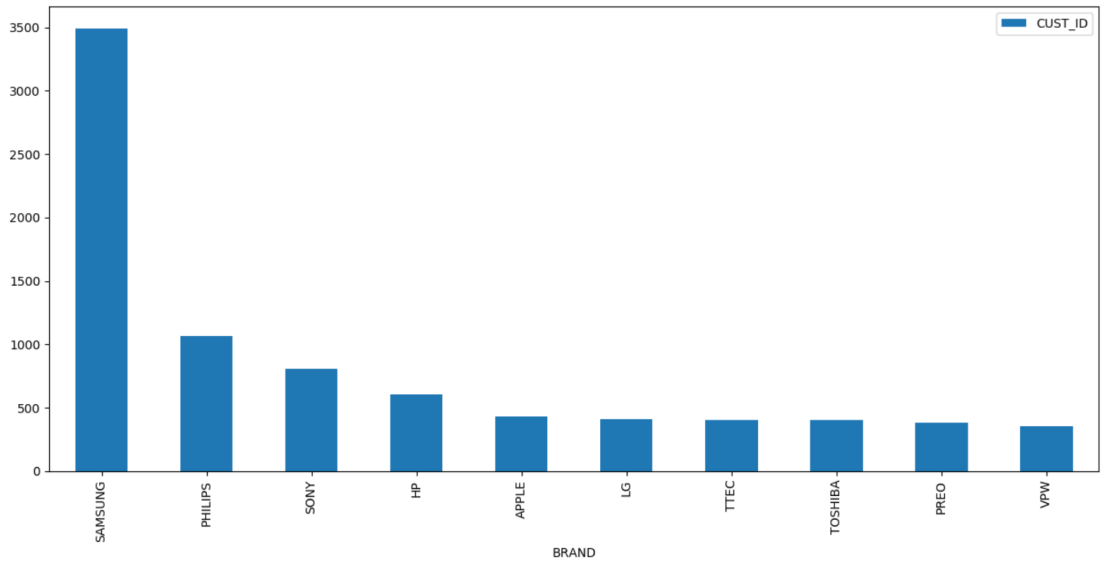
```
#Find unique Brands:
print(data[categorical]["BRAND"].describe())
unique_segment = data.groupby('BRAND')["CUST_ID"].nunique()
print("Unique Brands:\n",unique_segment)
```

There are 257 unique brands and some of them:

BRAND	
ACER	3
AEG	1
AIRTIES	5
ALCATEL	7
ALTUS	1
APPLE	435
ARAL	28
ARCON LONDON	3
ARNICA	5
ARZUM	64
ASELSAN	1
ASUS	84
AVEA	1
AXEN	2
BABYLISS	14
BANDRIDGE	1
BLACKBERRY	3
BOSCH	2
BRAUN	80
BRITA	3
CANON	34
CARS	1
CASE LOGIC	5
CASIO	4
CASPER	68
CELLY	7

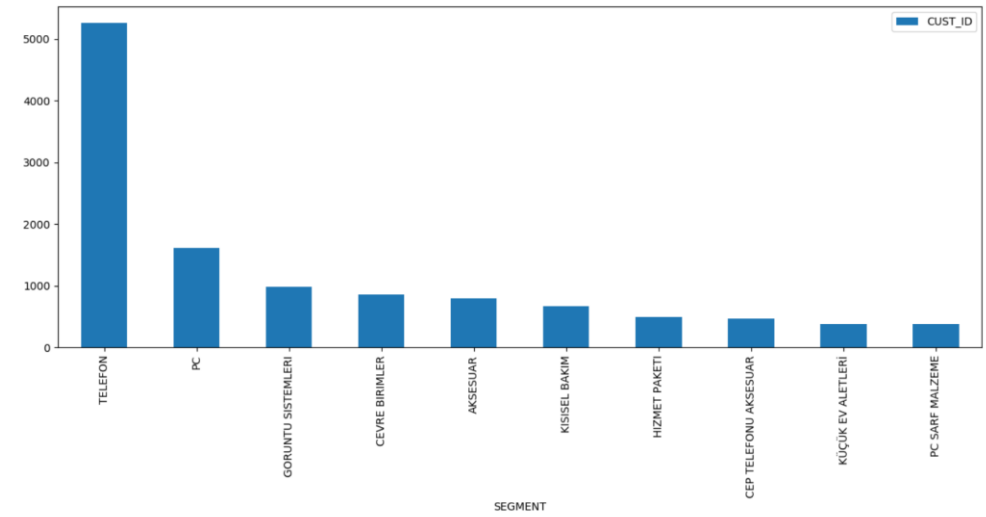
- Graph the most popular brands according to number of customer:

```
#Group Data according to SEGMENTS  and BRANDS and find descriptives:
grouped_brand= data[['CUST_ID','BRAND']].groupby(['BRAND'])
grouped_brand_cnt= grouped_brand.count()
grouped_brand_cnt=grouped_brand_cnt.sort_values(['CUST_ID'],ascending=False)
grouped_brand_cnt.head(10)

top_10_brand=grouped_brand_cnt.head(10)
top_10_brand=pd.DataFrame(top_10_brand)
top_10_brand.plot.bar()
```



- Graph the most popular segments according to number of customer:

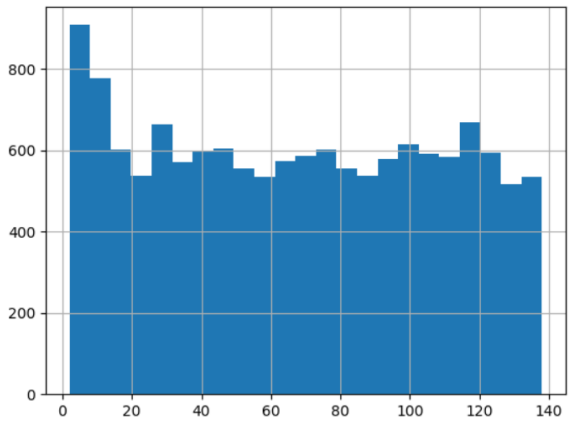
```
#Group Data according to SEGMENTS  and BRANDS and find descriptives:
grouped_segment= data[['CUST_ID','SEGMENT']].groupby(['SEGMENT'])
grouped_segment_cnt= grouped_segment.count()
grouped_segment_cnt=grouped_segment_cnt.sort_values(['CUST_ID'],ascending=Fa
lse)
print(grouped_segment_cnt.head(10))

top_10=grouped_segment_cnt.head(10)
top_10=pd.DataFrame(top_10)
top_10.plot.bar()
```

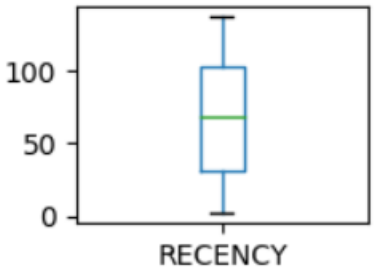


- Histogram of continuous columns:

  RECENCY:

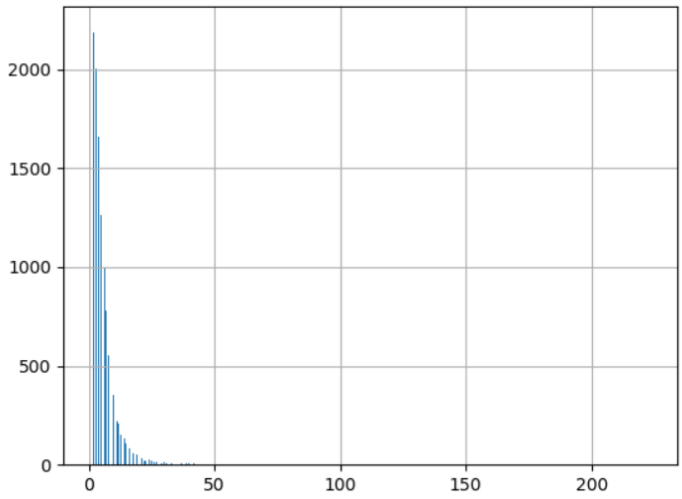
```
data["RECENCY"].hist()
data["RECENCY"].plot(kind='box', subplots=True, layout=(3,3), sharex=False, sharey=False)
```



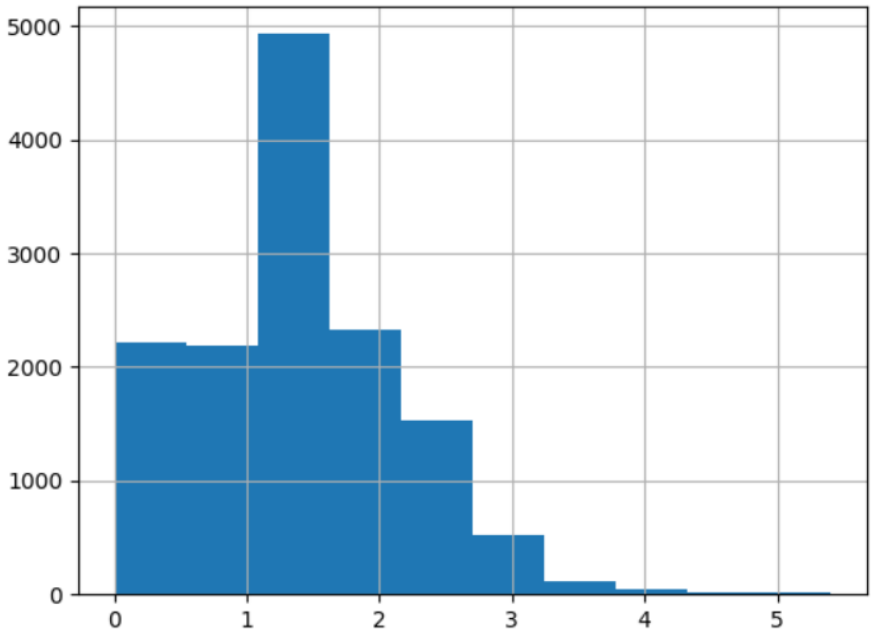


FREQUENCY:

```
data["FREQUENCY"].hist(bins="auto")
```

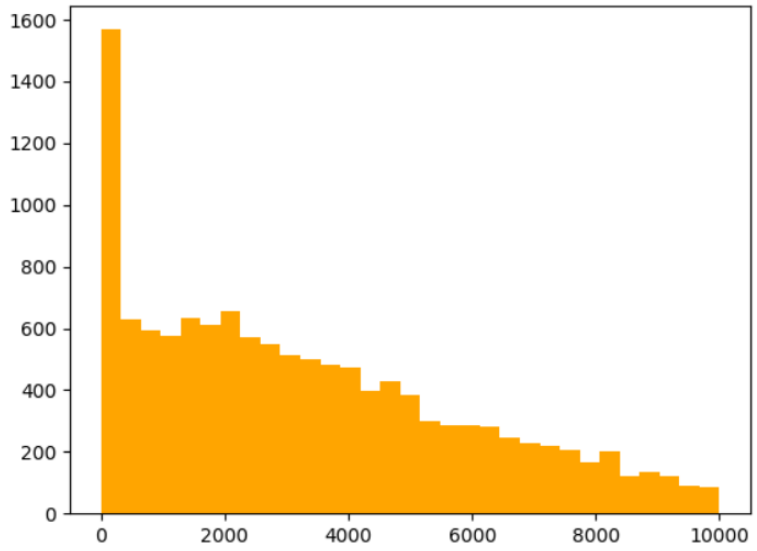



```
np.log(data["FREQUENCY"]).hist(bins=10)
```



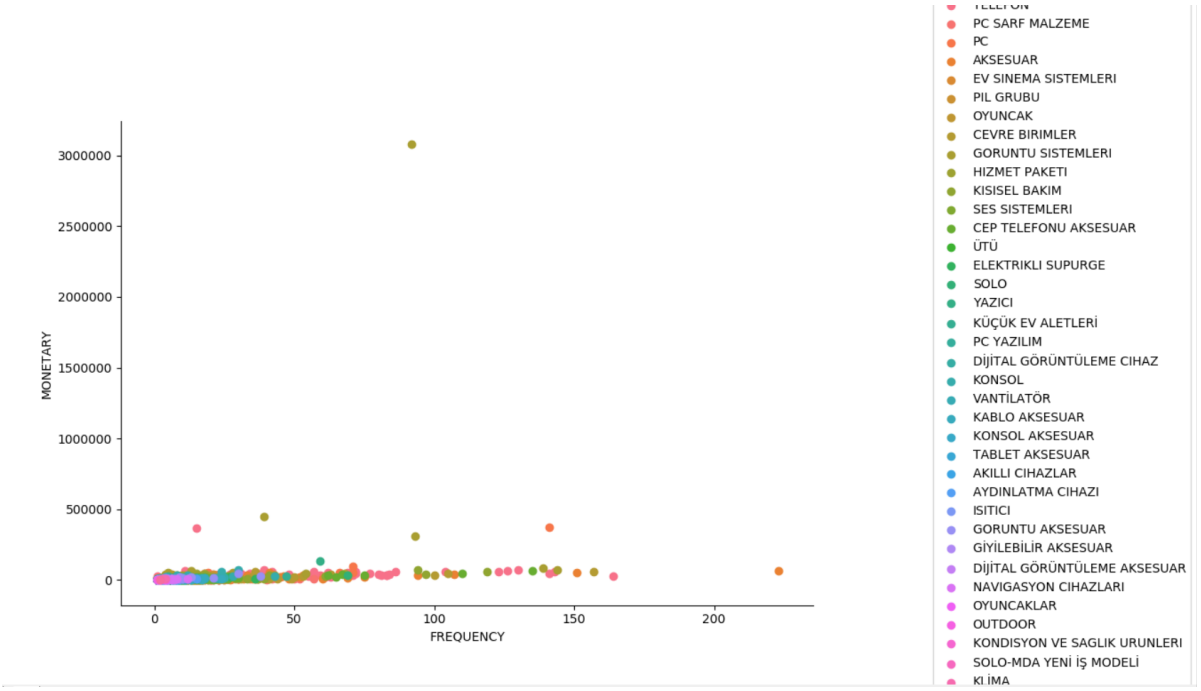
MONETARY:

```
plt.hist(data["MONETARY"], range=(0,10000),bins='auto', color=['orange'])
```



- Relationship between frequency, monetary and segment:

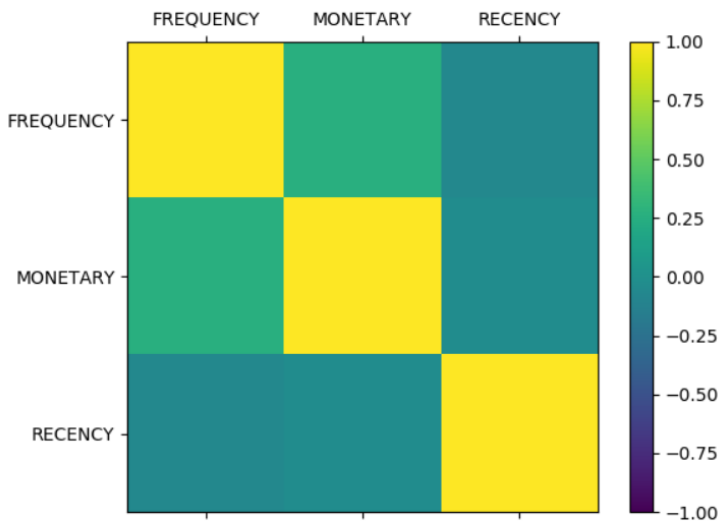
```
sns.FacetGrid(data, hue="SEGMENT", size=5) \
   .map(plt.scatter, "FREQUENCY", "MONETARY") \
   .add_legend()
```



As shown below; There are some outlier points and “GÖRÜNTÜ SİSTEMLERİ” has a marginal point.

- Correlation Matrix of these three columns:

```
names = ['FREQUENCY', 'MONETARY', 'RECENCY']
data_g = data[['FREQUENCY', 'MONETARY', 'RECENCY']]
correlations = data_g.corr()
# plot correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0,3,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names)
ax.set_yticklabels(names)
plt.show()
```



Due to correlation matrix; frequency and recency are negatively correlated as expected while frequency and monetary has less correlation. Additionally, monetary is highly correlated with recency.

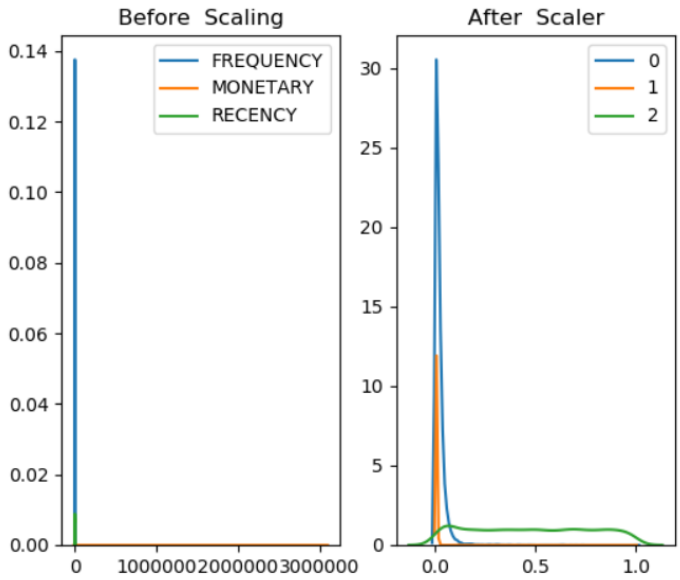
3. PRE - PROCESSING

While dataset have three numeric columns, exploratory data analysis show that these columns should be scaled. Although, there are a lot of methods to scale data; I have used the most popular methods from skilearn and examine the results for the best choice.

3.1. Data Scaling / Transformation

3.1.1. MinMax Scaler

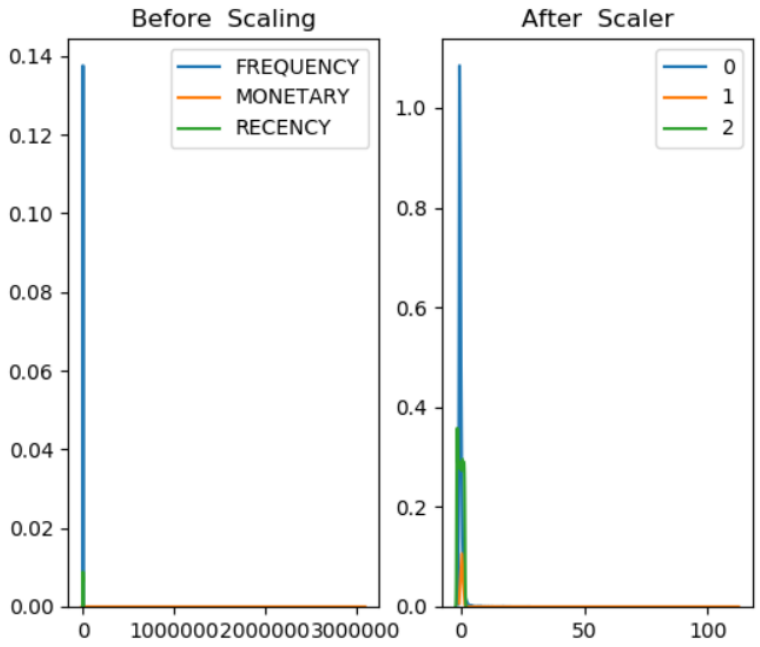
```
scaler = preprocessing.MinMaxScaler()
np_scaled = scaler.fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
df_normalized = pd.DataFrame(np_scaled)


fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))
ax1.set_title('Before  Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
ax2.set_title('After  Scaler')
sns.kdeplot(df_normalized[0], ax=ax2)
sns.kdeplot(df_normalized[1], ax=ax2)
sns.kdeplot(df_normalized[2], ax=ax2)
plt.show()
```



While MinMax Scaler is sensitive to outliers, it does not work on this dataset. RECENCY column seems not scaled very well.

3.1.2. StandardScaler

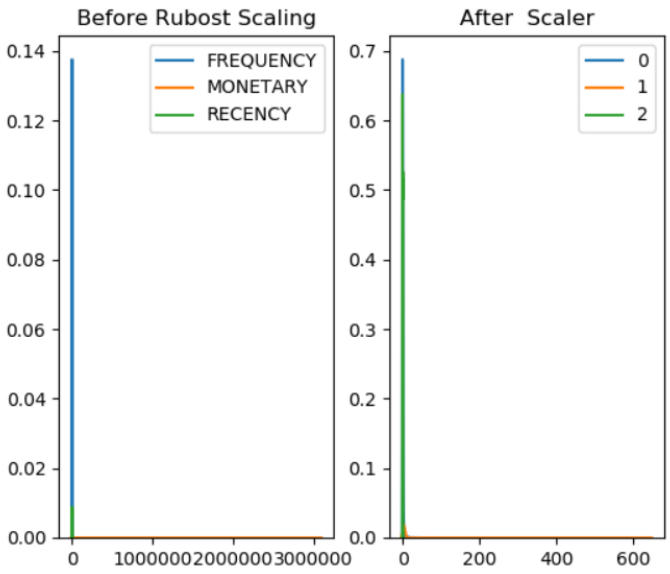
```
scaler = preprocessing.StandardScaler()
np_scaled = scaler.fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
df_normalized = pd.DataFrame(np_scaled)

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))
ax1.set_title('Before  Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
ax2.set_title('After  Scaler')
sns.kdeplot(df_normalized[0], ax=ax2)
sns.kdeplot(df_normalized[1], ax=ax2)
sns.kdeplot(df_normalized[2], ax=ax2)
plt.show()
```



While data is not normally distributed, Standard Scaler, also, does not work very well as above graph shown. Scale interval is wide but values are gathered at a narrow space. So, there are still outliers, especially on MONETARY column, that cannot be handled.

3.1.3. RubostScaler

```
scaler = preprocessing.RobustScaler(quantile_range=(25, 75))
np_scaled = scaler.fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
df_normalized = pd.DataFrame(np_scaled)

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))

ax1.set_title('Before Rubost Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
ax2.set_title('After  Scaler')
sns.kdeplot(df_normalized[0], ax=ax2)
sns.kdeplot(df_normalized[1], ax=ax2)
sns.kdeplot(df_normalized[2], ax=ax2)
plt.show()
```

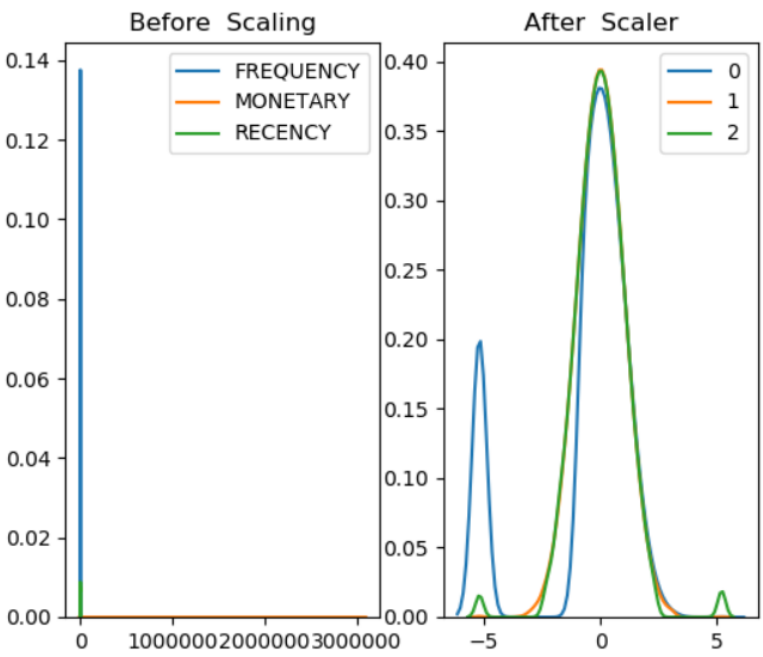


While rubost scaler is the best solution for outliers, it does not work on this dataset very well because the interval of values vary column by column, it cannot handle all outliers for each column.

3.1.4. QuantileTransformer

```
np_scaled = scaler.fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
df_normalized = pd.DataFrame(np_scaled)
 print(df_normalized.describe())


fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))
ax1.set_title('Before  Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
ax2.set_title('After  Scaler')
sns.kdeplot(df_normalized[0], ax=ax2)
sns.kdeplot(df_normalized[1], ax=ax2)
sns.kdeplot(df_normalized[2], ax=ax2)
plt.show()
```

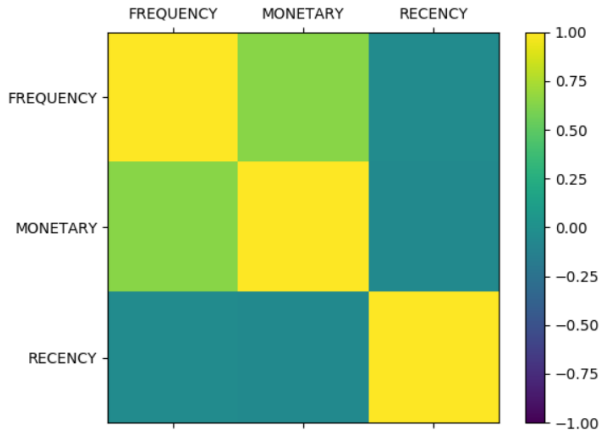
	0	1	2
count	13897.000000	13897.000000	13897.000000
mean	-0.583739	-0.000391	0.003921
std	2.132960	1.001151	1.131617
min	-5.199338	-5.199338	-5.199338
25%	-0.713599	-0.674418	-0.681594
50%	0.051460	-0.000006	0.006273
75%	0.743040	0.674648	0.678433
max	5.199338	5.199338	5.199338



QuantileTransformer gives the best result for now as shown above graph. This method transforms features using quantiles information. Prefer normal distribution as output of transformation because on modeling part, plan to use K-Means algorithm and K-Means does not work very well on uniform data. QuantileTransformer tends to spread out the most frequent values. It also reduces the impact of outliers. Additionally, transformation is applied on each feature independently. The cumulative density function of a feature is used to project the original values. Features values of new/unseen data that fall below or above the fitted range will be mapped to the bounds of the output distribution. Therefore, it works better than RubostScaler on this dataset.

However, this transform is non-linear. So, we should draw correlation matrix again after this transformation:

```
names = ['FREQUENCY', 'MONETARY', 'RECENCY']
data_g = df_normalized
correlations = data_g.corr()
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0,3,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names)
ax.set_yticklabels(names)
plt.show()
```



The result is better after transformation. The correlation between MONETARY and FREQUENCY has been increased while the other correlations are not affected by transformation so much.

To sum up, I decide to use scaled data that transformed by QuantileTransformer.

3.2. Cleaning

After scaling; data should be merged with new scaled columns and after cleaned.

3.2.1. Add Scaled Columns

- Rename transformed columns:

```
#rename transformed columns:
df_normalized.rename(columns={0:"FREQUENCY_SCALED",1:"MONETARY_SCALED",2:"RECENCY_SCALED"},inplac
e=True)
print(df_normalized.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13897 entries, 0 to 13896
Data columns (total 3 columns):
FREQUENCY_SCALED    13897 non-null float64
MONETARY_SCALED     13897 non-null float64
RECENCY_SCALED      13897 non-null float64
dtypes: float64(3)
memory usage: 325.8 KB
None
```

- Concatenate scaled data and original dataset:

```
#concat scaled data nad other categorical columns:
q_transformed_data= pd.concat([df_normalized,data],axis=1)
print(q_transformed_data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13897 entries, 0 to 13896
Data columns (total 9 columns):
FREQUENCY_SCALED    13897 non-null float64
MONETARY_SCALED     13897 non-null float64
RECENCY_SCALED      13897 non-null float64
CUST_ID             13897 non-null int64
RECENCY             13897 non-null int64
FREQUENCY           13897 non-null int64
MONETARY            13897 non-null int64
SEGMENT             13897 non-null object
BRAND               13897 non-null object
dtypes: float64(3), int64(4), object(2)
memory usage: 977.2+ KB
None
```

- Drop unnecessary columns:

```
del q_transformed_data["CUST_ID"]
del q_transformed_data["MONETARY"]
del q_transformed_data["FREQUENCY"]
del q_transformed_data["RECENCY"]
```

- Describe new dataset:

```
print(q_transformed_data.info())
print(q_transformed_data.describe())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13897 entries, 0 to 13896
Data columns (total 5 columns):
FREQUENCY_SCALED    13897 non-null float64
MONETARY_SCALED     13897 non-null float64
RECENCY_SCALED      13897 non-null float64
SEGMENT             13897 non-null object
BRAND               13897 non-null object
dtypes: float64(3), object(2)
memory usage: 542.9+ KB
None
       FREQUENCY_SCALED  MONETARY_SCALED  RECENCY_SCALED
count      13897.000000     13897.000000    13897.000000
mean          -0.583739        -0.000391        0.003921
std            2.132960         1.001151        1.131617
min           -5.199338        -5.199338       -5.199338
25%           -0.713599        -0.674418       -0.681594
50%            0.051460        -0.000006        0.006273
75%            0.743040         0.674648        0.678433
max            5.199338         5.199338        5.199338
```

3.2.2. Create Dummy Columns from SEGMENT Column

Categorical columns should be converted into numeric columns for clustering. While BRAND column has more than 200 brand, it is better to drop this column.

- Create dummy columns from SEGMENT column:

```
lb_style = LabelBinarizer()
lb_results = lb_style.fit_transform(q_transformed_data["SEGMENT"])
lb_results=pd.DataFrame(lb_results, columns=lb_style.classes_)
```

- Concatenate new dummy columns to transformed data:

```
q_transformed_data= pd.concat([q_transformed_data,lb_results],axis=1)
print(q_transformed_data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13897 entries, 0 to 13896
Data columns (total 80 columns):
FREQUENCY_SCALED                13897 non-null float64
MONETARY_SCALED                 13897 non-null float64
RECENCY_SCALED                  13897 non-null float64
CUST_ID                         13897 non-null int64
AKILLI CIHAZLAR                 13897 non-null int32
AKSESUAR                        13897 non-null int32
AYDINLATMA CIHAZI               13897 non-null int32
CEP TELEFONU AKSESUAR           13897 non-null int32
CEVRE BIRIMLER                  13897 non-null int32
DİJİTAL GÖRÜNTÜLEME AKSESUAR    13897 non-null int32
DİJİTAL GÖRÜNTÜLEME CIHAZ       13897 non-null int32
ELEKTRIKLI SUPURGE              13897 non-null int32
EV SINEMA SISTEMLERI            13897 non-null int32
GORUNTU AKSESUAR                13897 non-null int32
GORUNTU SISTEMLERI              13897 non-null int32
GİYİLEBİLİR AKSESUAR            13897 non-null int32
```

- Drop unnecessary categorical columns:

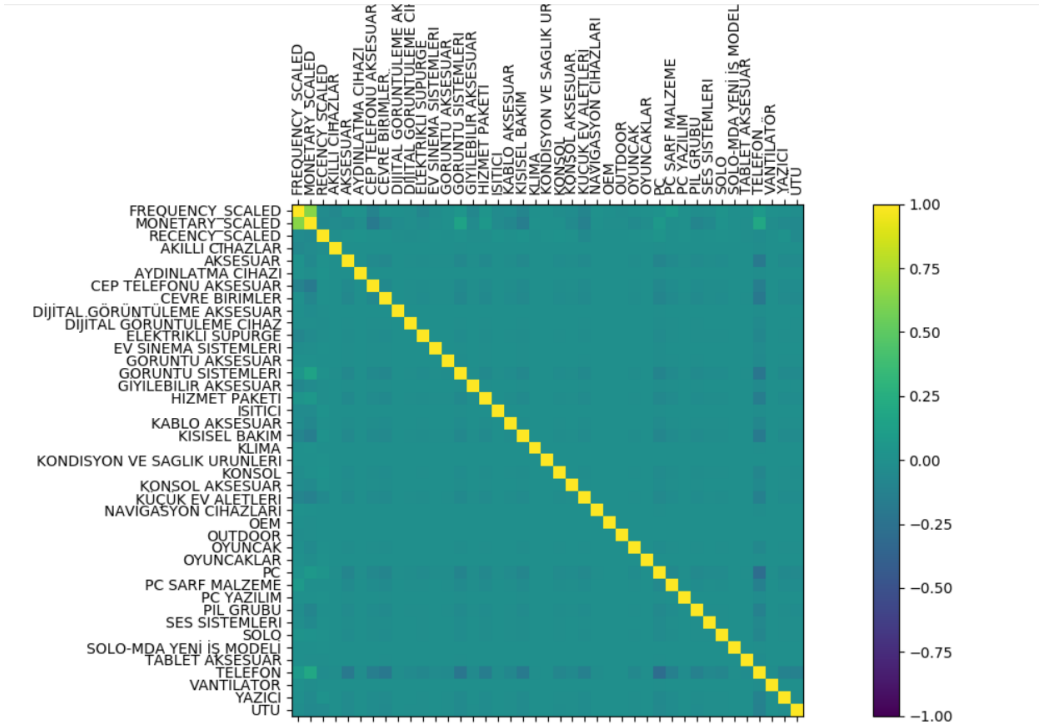
```
del q_transformed_data["SEGMENT"]
del q_transformed_data["BRAND"]
```

- Check dummy column data:

```
>>> print(data.loc[0])
CUST_ID             1
RECENCY             2
FREQUENCY           4
MONETARY         2032
SEGMENT       TELEFON
BRAND            ASUS
Name: 0, dtype: object
>>> print(q_transformed_data.loc[0]["TELEFON"])
1.0
```

- Correlation after dummy columns:

```
names = q_transformed_data.columns #['FREQUENCY', 'MONETARY', 'RECENCY']
data_g = q_transformed_data #data[['FREQUENCY', 'MONETARY', 'RECENCY']]
correlations = data_g.corr()
# plot correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0,41,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names,rotation='vertical')
ax.set_yticklabels(names) # fontdict={'verticalalignment': 'baseline'})
plt.show()
```



4. MODELING

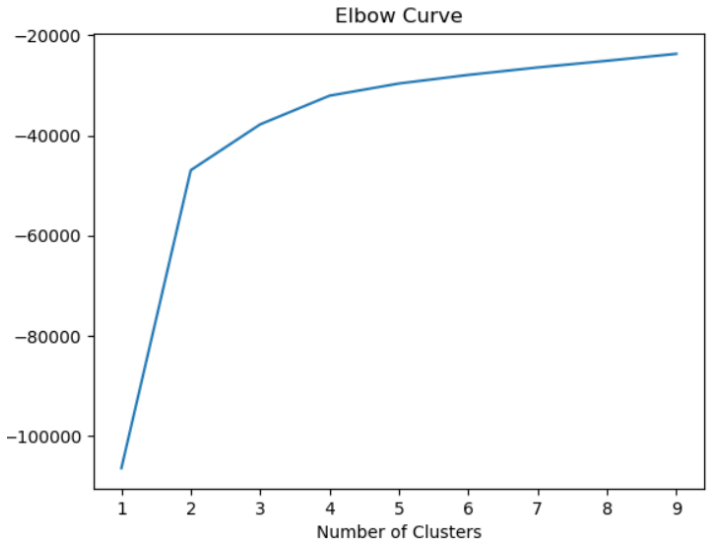
4.1. K-MEANS

Deciding number of cluster is the biggest challenge of K-Means. So, first step of modeling with K-Means is searching a reasonable number of cluster.

K-Means with Row Data

- Elbow rule: Specifically, devising a range from 1 to 10 (which represents number of clusters), and score variable denotes the percentage of variance explained by the number of clusters.

```
Nc = range(1, 10)
kmeans = [KMeans(n_clusters=i) for i in Nc]
kmeans
score = [kmeans[i].fit(q_transformed_data).score(q_transformed_data) for i in
range(len(kmeans))]
score
pl.plot(Nc,score)
pl.xlabel('Number of Clusters')
pl.ylabel('Score')
pl.title('Elbow Curve')
pl.show()
```



Elbow curve show that 4 is the best choice for number of cluster to give as input to K-Means.
Scores:

[-106375.12462799408, -46932.804475808705, -37789.3825187339, -**32066.946890418985**, -29636.06403809609, -27921.31857626944, -26437.228011941337, -25058.737182608787, -23736.160902535194]

- Silhouette Score: Silhouette score is the best choice to measure clustering algorithms while data has no label. So, comparing silhouette score for different number of cluster will show the most reasonable number on K-Means. Firstly, I defined a function which apply K-Means algorithm for 3,4,5,6,7,8 clusters and return two different list hold silhouette scores (Euclidean and cosine) for each number of cluster.

```
def k_means(q_transformed_data):
    num_clusters = [3,4,5,6,7,8]
    sil_dic={}
    sil_ecd_dic={}
    for i in num_clusters:
        print("************************* n_cluster= %d ************************" %i)
        kmeans = KMeans(init='k-means++',n_clusters= i, n_jobs= -1)
        t0 = time()
        kmeans.fit(q_transformed_data)
        t1 = time() - t0
        print("K-means took: %.3f ", t1)

        df = pd.DataFrame(kmeans.cluster_centers_)
        df['count'] = pd.Series(kmeans.labels_).value_counts()  # Number of each clusters
        print("Number of elements for each cluster: \t")
        print(df['count'])
        #print("K-means took:\t%.3f "% t1, "sec")
        silhouette_cosine= metrics.silhouette_score(q_transformed_data,kmeans.labels_,
metric='cosine', sample_size=q_transformed_data.shape[0])
        silhouette_ecludiene= metrics.silhouette_score(q_transformed_data,kmeans.labels_,
metric='euclidean', sample_size=q_transformed_data.shape[0])
        print("silhouette score %.3f" %silhouette_ecludiene)
      ch_score=metrics.calinski_harabaz_score(q_transformed_data,kmeans_output.labels_)
      print("silhouette score %.3f" %ch_score)
      sil_dic[i]=silhouette_cosine
      sil_ecd_dic[i]=silhouette_ecludiene
      print(sil_dic)
       print(sil_ecd_dic)
      return sil_dic,sil_ecd_dic


sil_dic,sil_ecd_dic = k_means(q_transformed_data)
```

Output of this function print some values for each cluster:

```
************************* n_cluster= 3 ************************
Number of elements for each cluster:
0    6544
1    2211
2    5142
Name: count, dtype: int64
silhouette score 0.272
************************* n_cluster= 4 ************************
Number of elements for each cluster:
0    2211
1    3592
2    4095
3    3999
Name: count, dtype: int64
silhouette score 0.261
************************* n_cluster= 5 ************************
Number of elements for each cluster:
0    3101
1    2211
2    3739
3    2378
4    2468
Name: count, dtype: int64
silhouette score 0.231
************************* n_cluster= 6 ************************
Number of elements for each cluster:
0    1496
1    2211
2    2509
3    1443
4    2996
5    3242
Name: count, dtype: int64
silhouette score 0.223
```

Due to this output 4 is the best for number of cluster but silhouette score is really low. So, I have decided to apply PCA before K-Means clustering.

- When we look at evaluation metrics for model; we should consider K-Means score, silhouette score and Calinski Harabaz Score while labels are unknown on this dataset.

- Summary of K-Means on raw data with 4 Clusters:

K-Means Score	Silhouette Score	Calinski harabaz Score	Time (sec)
-32066.9468	0.261	10731.339	3.263

K-Means with Transformed Data with PCA

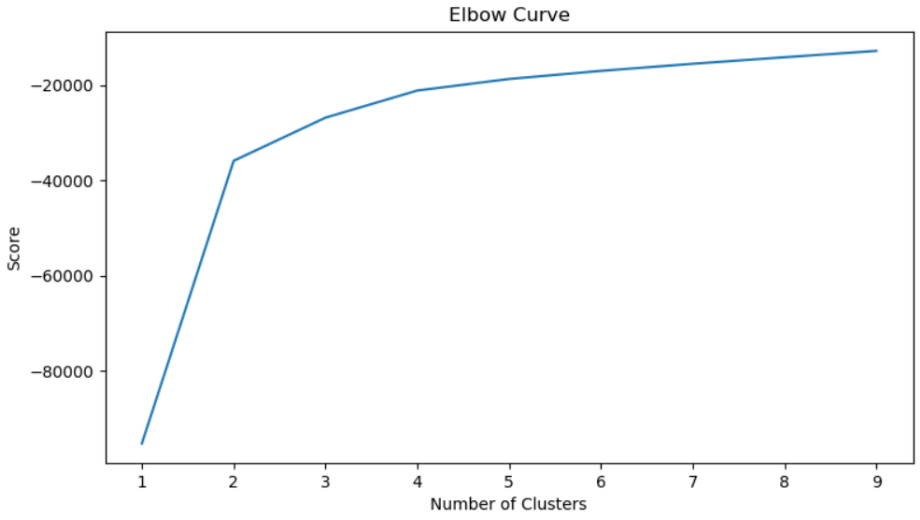
- Apply PCA to data and show variance_ratio:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=3)
reduced_data = pca.fit_transform(q_transformed_data)
#reduced_data = PCA(n_components=2).fit_transform(q_transformed_data)
print(pca.explained_variance_ratio_)
```

```
...
[0.65649663 0.16722715 0.07189175]
```

Variance_ratio show that PCA with 3 principal components explains around 88% of data. So, reduced_data can be used instead of raw data.

- Draw elbow curve to select the best number of cluster before applying K-Means:



Elbow curve shows that 4 is, again, the best number of cluster for reduced data but in any case, silhouette scores should be checked.

Scores:

[-95271.21373482057, -35854.63612055738, -26788.98571111149, -**21089.175232732956**, -18657.54727241687, -16956.994016566263, -15466.794471146506, -14086.681914619552, -12756.685751851966]

- Call defined k_means function with reduced_data:

```
k_means(reduced_data)
```

The list of silhouette scores due to number of cluster given to K-means algorithm:

{3: 0.3582008473160527, 4: 0.42495496148883277, 5: 0.38021246272579057, 6: 0.2855798511409984, 7: 0.21142169304656414, 8: 0.21937911926792034}

Silhouette score also shown that 4 is the best number of cluster for K-Means algorithm with reduced data with PCA.

```
************************ n_cluster= 3 ************************
Number of elements for each cluster:
0    6487
1    2211
2    5199
Name: count, dtype: int64
silhouette score 0.346
************************ n_cluster= 4 ************************
Number of elements for each cluster:
0    3998
1    2211
2    3636
3    4052
Name: count, dtype: int64
silhouette score 0.345
************************ n_cluster= 5 ************************
Number of elements for each cluster:
0    2916
1    2211
2    3660
3    2646
4    2464
Name: count, dtype: int64
silhouette score 0.318
************************ n_cluster= 6 ************************
Number of elements for each cluster:
0    3250
1    2211
2    1479
3    1445
4    2494
5    3018
```

- Apply K-Means and show metrics:

```
kmeans = KMeans(init='k-means++',n_clusters= 4, n_jobs= -1)
kmeansoutput=kmeans.fit(reduced_data)

df = pd.DataFrame(kmeans.cluster_centers_)
df['count'] = pd.Series(kmeans.labels_).value_counts()  # Number of each clusters
print("Number of elements for each cluster: \t")
print(df['count'])
```

```
Number of elements for each cluster:
0    4057
1    2211
2    4038
3    3591
```

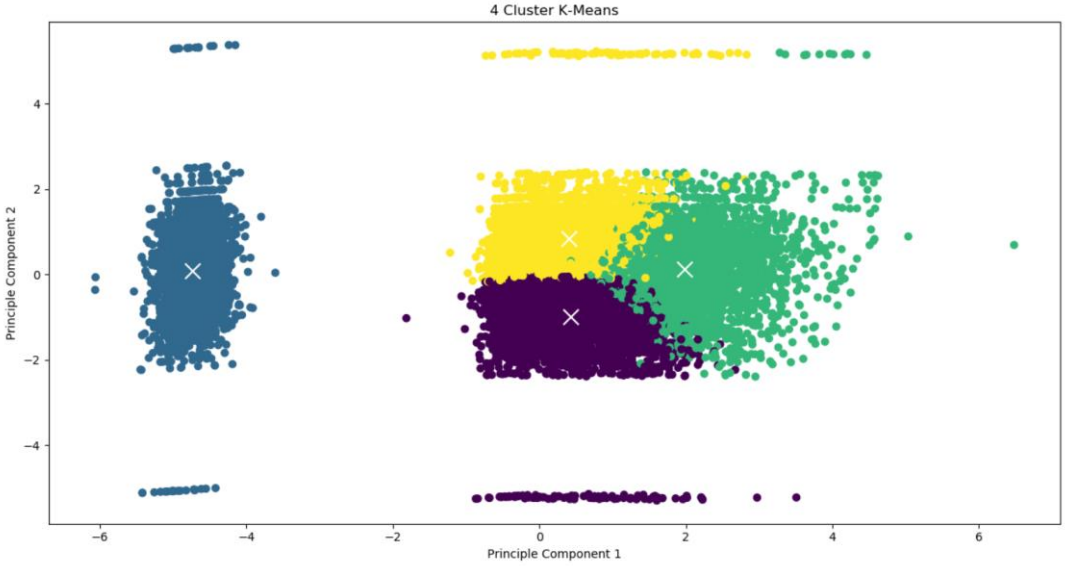
As shown on output; there are 4 clusters.

```
silhouette_cosine= metrics.silhouette_score(reduced_data,kmeans.labels_, metric='cosine',
sample_size=reduced_data.shape[0])

print(silhouette_cosine)
```

- **Silhouette score is 0.42672 for reduced data with 4 clusters while it is 0.26 for row data where a higher Silhouette Coefficient score relates to a model with better defined clusters. The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.**

- Draw cluster graph due to labels of K-Means:
  While principle component 3 variance_Ratio is 0.071, the graph can be drawn with other components have higher variance ratio.

```
pl.figure('4 Cluster K-Means')
pl.scatter(reduced_data[:, 0],reduced_data[:, 1], c=kmeansoutput.labels_)
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)
pl.xlabel('Principle Component 1')
pl.ylabel('Principle Component 2')
pl.title('4 Cluster K-Means')
pl.show()
```



The scatter graph shows that although there are marginal points for each cluster, K-Means after PCA give better results.

Compare two K-Means

Comparison table for two K-Means:

K-Means with 4 Clusters				
	Score	**Silhouette Score**	**Calinski-Harabasz Score**	**Run Time(sec)**
Raw Data	-32066.9469	0.2610	10731.339	3.263
After PCA	-21089.1752	0.4267	16289.3910	3.1160

- **Comparing Score:** K-Means' score() function returns opposite of the value of X on the K-means objective where the *objective* of our minimization is that distance (sum of all distances between datapoints and closest cluster center). So, K-Means with reduced data (generated after PCA) is better due to score, because we have less sum of distances on this model. It means generated clusters are more dense after PCA.

- **Comparing Silhouette Score:** The silhouette score is bounded between -1 for incorrect clustering and +1 for highly dense clustering. Scores around zero indicate overlapping clusters.
The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster. When we compare two silhouette score, after PCA silhouette score is 0.42. Therefore, PCA affects clusters in a positive way and causes create more dense and well separated clusters.

- **Comparing Calinski-Harabasz Score:** The Calinski-Harabasz score is given as the ratio of the between-clusters dispersion where a higher Calinski-Harabasz score relates

to a model with better defined clusters. After PCA, we have better defined clusters due to Calinski Harabasz Score.

- **Comparing Run Time:** As expected, runtime decreases after PCA because on raw data, there are 13897 rows and 41 columns to process. After PCA, K-Means input data has 3 columns with same row count.

While we compare two K-Means algorithms before and after PCA, K-Means works on dimensional reduced data. Main reason of that is K-Means' algorithmic logic. This algorithm is based on calculating distance between data points and decide center of clusters due to this distance calculations. So, while data points have so many dimension; calculating distance is more complex and open to fault. Additionally, more dimension may cause noisy and sparse data. PCA convert this multi-dimensional representation into less directional plane with little information loss and give better model results. Because of this, K-Means works better on dimensional reduced data by PCA. In conclusion, reduced data should be used as input of other clustering methods.

4.2. AGGLOMERATIVE CLUSTERING

The Agglomerative Clustering performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together.

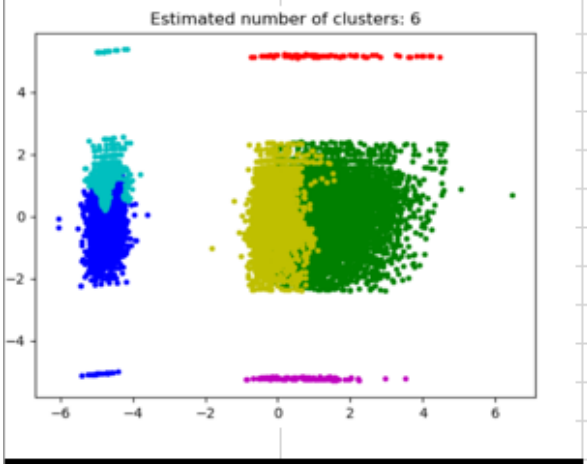
The linkage criteria determine the metric used for the merge strategy: ward, complete and average. So, all linkage type tried on reduced dataset where affinity is Euclidean, l2 and cosine but affinity does not affect the results. So, just different output due to linkage type are examined.

Main Code Chunk:

```
from sklearn.cluster import AgglomerativeClustering

X=reduced_data

clt = AgglomerativeClustering(linkage='average',
                              affinity='euclidean',
                              n_clusters=4)
t0 = time()
model = clt.fit(X)
t1 = time() - t0
labels = clt.labels_
print(labels)
df = pd.DataFrame(reduced_data,labels)
df['count'] = pd.Series(labels).value_counts()
#print("Number of element due to Clusters:\n",df['count'][1][1])
print("Runtime for Agglomerative: ",t1)
print("Silhouette Coefficient: %0.3f" % metrics.silhouette_score(X, labels))
print("CH Coefficient: %0.3f" % metrics.calinski_harabaz_score(reduced_data,labels))

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)


# Plot result
import matplotlib.pyplot as plt
from itertools import cycle


plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

On this code chunk, linkage and n_clusters parameters are changed for each step and the best results are as follow:

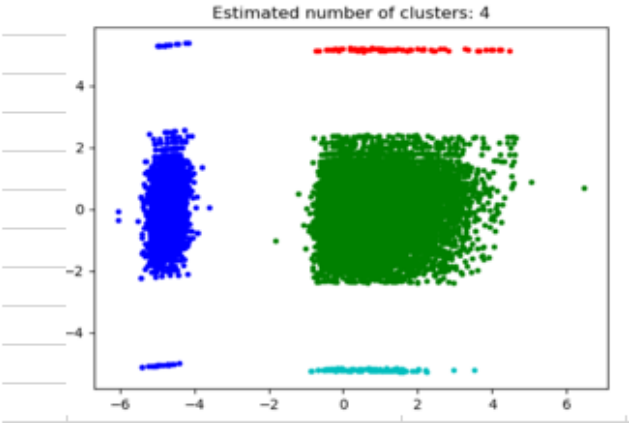
linkage:complete	n_clusters:6
Silhouette Score	0.237
Calinski-Harabasz Sc	8430.679
Run Time(sec)	7.970



linkage:ward	n_clusters:4
Silhouette Score	0.291
Calinski-Harabasz Score	13907.803
Run Time(sec)	6.640

linkage:complete	n_clusters:4
Silhouette Score	0.231
Calinski-Harabasz Sc	10955.899
Run Time(sec)	7.370

linkage:average	n_clusters:4
Silhouette Score	0.600
Calinski-Harabasz Score	9502.890
Run Time(sec)	7.098



As shown above; agglomerative clustering gives the best result when linkage=average and n_clusters=4. The clusters are mainly split, there is no overlap and silhouette score is 0.6 with reasonable runtime.

As mentioned before; the linkage criteria determine the metric used for the merge strategy. Ward minimizes the sum of squared differences within all clusters; because of this, ward linkage split the big cluster into three different clusters when n_clusters=4 because agglomerative clustering tries to minimize until generated

clusters match n_clusters parameter. So, ward could not minimize the sum of squared differences while satisfying n_clusters on this dataset. Secondly, complete linkage minimizes the maximum distance between observations of pairs of clusters. Same reason with ward; when trying to satisfy n_clusters parameter, complete linkage split one big cluster into two clusters and cause overlapped. That's because silhouette score is more close to 0. Finally, when linkage=average and n_clusters=4; silhouette score is 0.6 and the graph of this result is the best; clusters are not overlapped, all marginal points are covered by reasonable cluster. In conclusion, when compare the other clustering methods, the result of this combination of agglomerative clustering should be used.

4.3. MEAN-SHIFT

Mean shift clustering aims to discover "blobs" in a smooth density of samples. It is a centroid-based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

Mean-shift algorithm decides the number of clusters, unlike K-Means and agglomerative clustering, based on bandwidth parameter (which dictates the size of the region to search through.). Bandwidth can be set manually or can be estimated using estimate_bandwidth function.

Main Code Chunk:

```
from sklearn.cluster import MeanShift, estimate_bandwidth

X=reduced_data
bandwidth = estimate_bandwidth(reduced_data, quantile=0.1, n_samples=50)
ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
t0 = time()
meanshift_output=ms.fit(X)
t1 = time() - t0
labels = ms.labels_
cluster_centers = ms.cluster_centers_
df = pd.DataFrame(cluster_centers)
df['count'] = pd.Series(labels).value_counts()
print("Number of element due to Clusters:\n",df['count'])
print("Runtime for Mean-Shift: ",t1)
print("Silhouette Coefficient: %0.3f" % metrics.silhouette_score(X, labels))
print("CH Coefficient: %0.3f" % metrics.calinski_harabaz_score(reduced_data,labels))

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)
# Plot result
import matplotlib.pyplot as plt
from itertools import cycle
plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

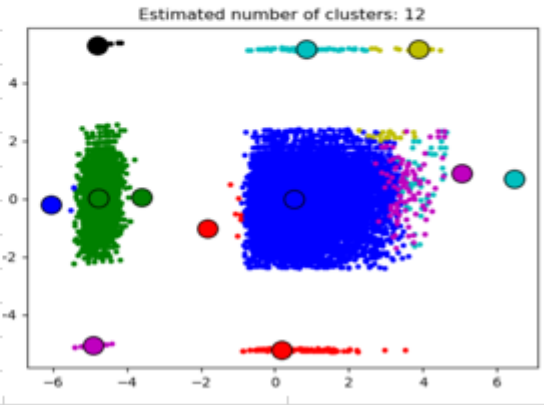
On this code chunk, quantile and n_samples parameters of estimate_bandwidth are changed for each step and the best results are as follow:

quantile=0.2	n_samples=2000
Estimated #cluster	6
Silhouette Score	0.589
Calinski-Harabasz Score	5942.088
Runtime (sec)	0.2835

quantile=0.2	n_samples=all samples
Estimated #cluster	5
Silhouette Score	0.594
Calinski-Harabasz Score	7292.605
Runtime (sec)	0.262

quantile=0.3	n_samples=50
Estimated #cluster	7
Silhouette Score	0.445
Calinski-Harabasz Score	4974.78
Runtime (sec)	0.3109

quantile=0.1	n_samples=50
Estimated #cluster	12
Silhouette Score	0.293
Calinski-Harabasz Score	2978.954
Runtime (sec)	1.42



As shown above; when n_samples=50, bandwidth generates more clusters but they are not meaningful. When quantile=0.2; the results are good for this dataset but examining evaluation metrics shows that quantile=0.2 and n_Sample=all samples gives the best results. It has better silhouette and CH score, runtime is less. . In conclusion, when compare the other clustering methods, the result of this combination of mean-shift should be used.

5. CONCLUSION

On this analysis, there are 3 main processes as exploratory data analysis, pre-processing and modeling. The customer dataset has 13897 rows which includes frequency, recency, monetary, favorite product category and brand information of customers.

On exploratory data analysis, each column is examined and graphs shows distribution of continuous columns. Segment (product category column) has 38 and brand column has more than 200 different value.

On pre-processing part, continuous variables are transformed with Quantile Transformer after trial of different scaling methods. Additionally, clustering methods that used on this project does not except categorical columns. Nevertheless, BRAND column cannot be converted into dummy column with 200+ values. So, before modeling; brand column is dropped. However, SEGMENT column is appropriate to create dummy column. To sum up; on this part, continuous columns are scaled with the best scaler for data structure and create new 38 columns from SEGMENT column. After all, data is appropriate with 40+ columns to modeling part.

On modeling part, firstly K-Means is applied on raw data with different number of parameter. Although reasonable number of cluster is 4 when comparing the different number of cluster, silhouette score of that model is 0.261 for this algorithm. Thence, PCA is applied to dataset for dimensionality reduction. After PCA, reduced data has 3 principal components with %88 total variance ratio. Secondly, K-Means algorithm is applied on reduced data and silhouette score rises to 0.4267. While we compare two K-Means algorithms before and after PCA, K-Means works on dimensional reduced data. Main reason of that is K-Means' algorithmic logic. This algorithm is based on calculating distance between data points and decide center of clusters due to this distance calculations. So, while data points have so many dimension; calculating distance is more complex and open to fault. Additionally, more dimension may cause noisy and sparse data. PCA convert this multi-dimensional representation of data into less directional plane with little information loss and give better model results. In conclusion, reduced data should be used as input of other clustering methods.

Thirdly, a hierarchical clustering method is applied on reduced data, agglomerative clustering. After trials with different linkage and n_clusters parameter values, the best result is generated by the model that have linkage='average' and n_clusters=4. K-Means has already told 4 is reasonable as number of cluster for reduced data and linkage average minimizes the average of the distances between all observations of pairs of clusters. That's why silhouette score of this model is 0.600.

The last algorithm is Mean-Shift applied on reduced dataset. Mean-Shift does not need an initial number of cluster but it need bandwidth. Estimate_bandwidth function is used to calculate the best bandwidth for reduced data. After trials with different quantile and n_sample combination, the best result generated by the model that have

quantile=0.2, n_sample = (number of all samples) with silhouette score 0.594. Additionally, Mean-Shift estimates 5 clusters, unlike K-Means and agglomerative clustering gives the best results with 4 clusters. When we compare these three algorithms due to evaluation metrics, K-Means fails on reduced data. Depends on business logic, Agglomerative and mean-Shift are acceptable solutions for this dataset. If we want to divide our customers into 4 segments due to their behavior; agglomerative clustering should be applied. It takes more times than Mean-Shift but gives more Calinski-Harabasz Score while silhouette scores are so close to each other for each model. In contrast, if we want to divide our costumers into 5 segments, Mean-Shift is the best model for reduced data. Runtime of Mean-Shift is 0.262 sec while Calinski-Harabasz Score is less. Additionally, 5 clusters may include some exclusive customers which cannot be recognizable by other models.

In conclusion, although raw data needs some transformation before modeling; agglomerative clustering and Mean-Shift are convenient algorithms for transformed data.

	K-Means	Agglomerative	Mean-Shift
Number of Clusters	4	4	5
Silhouette Score	0.4267	0.600	0.594
Calinski-Harabasz Score	16289.3910	9502.890	7292.605
Run Time(sec)	3.1160	7.098	0.262

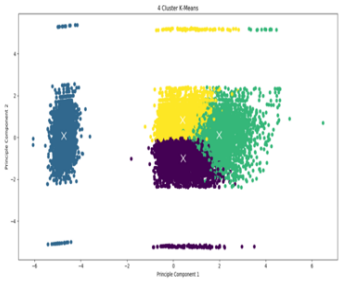


Figure: Comparison Table for three clustering models.

References

[1] Sara Dolnicar. Using cluster analysis for market segmentation - typical misconceptions, established methodological weaknesses and some recommendations for improvement. *University of Wollongong, 2003.*

[2] Konstantinos Tsiptsis, Antonios Chorianopoulos. Data Mining Techniques in CRM: Inside Customer Segmentation. *Wiley, 2009.*

[3] Houle1, Michael. Kriegel2, Hans-Peter. Kröger, Peer. "Can Shared-Neighbor Distances Defeat the Curse of Dimensionality?" *Germany*, 2010.

[4] Kimberley Coffey. *K-Means clustering for customer segmentation. http://www.kimberlycoffey.com/blog/2016/8/k-means-clustering-for-customer-segmentation (accessed 15 May 2018).*

[5] Clustering Methods. http://scikit-learn.org/stable/modules/clustering.html (accessed 01 June 2018).

[6] Model Metrics. http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics (accessed 01 June 2018).

[7] Mean Shift. http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics (accessed 01 June 2018).

[8] Mean Shift. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn-cluster-agglomerativeclustering (accessed 01 June 2018).

[9] Vincent Zhang. RFM Analysis for Air Company. https://www.kaggle.com/vinzzhang/rfm-model-for-customer-value-of-air-company (accessed 01 June 2018).

Appendix

```
import numpy as np
import pandas as pd
from numpy.ma.core import sort
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans  # to use the kmeans clustering algorithm
from sklearn.preprocessing import scale
from sklearn import preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import minmax_scale
from sklearn.preprocessing import MaxAbsScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing.data import QuantileTransformer
from sklearn.preprocessing import LabelBinarizer
import pylab as pl
from sklearn import metrics


#Read Data:
filename = 'data_full2.csv'
data = pd.read_csv(filename)
data.info()  #13897
print(data.describe())
data.columns  #['CUST_ID', 'RECENCY', 'FREQUENCY', 'MONETARY', 'SEGMENT', 'BRAND']

#check data:
data.head()
data.shape

#Categorical columns describe:
categorical = data.dtypes[data.dtypes == "object"].index
print(data[categorical].describe())


#Find unique SEGMENTS:
categorical = data.dtypes[data.dtypes == "object"].index
print(categorical)
print(data[categorical]["SEGMENT"].describe())
unique_segment = data.groupby('SEGMENT')["CUST_ID"].nunique()
print("Unique Segments:\n",unique_segment)

#Find unique Brands:
print(data[categorical]["BRAND"].describe())
unique_segment = data.groupby('BRAND')["CUST_ID"].nunique()
print("Unique Brands:\n",unique_segment)


#nullity check:
print("Nullity Check:\n",pd.isnull(data).any())
print("Sum of null values for each column:\n",pd.isnull(data).sum())


#Cleaning segment data:

sb_index=np.where(data["SEGMENT"]=='KONSOL OYUN')
#print(sb_index)
for i in sb_index:
    data.loc[i,"SEGMENT"]='KONSOL'
     #print(data.loc[i]["SEGMENT"])

sb_index=np.where(data["SEGMENT"]=='PC KOMPONENT')
#print(sb_index)
for i in sb_index:
    #print(data.loc[i]["SEGMENT"])
    #data.loc[i]["SEGMENT"]='PC'
    data.loc[i,"SEGMENT"]='PC'
    #print(data.loc[i]["SEGMENT"])

sb_index=np.where(data["SEGMENT"]=='SOLO-MDA Yeni İş Modeli')
#print(sb_index)
for i in sb_index:
    data.loc[i,"SEGMENT"]='SOLO'

print(data[categorical]["SEGMENT"].describe())
unique_segment = data.groupby('SEGMENT')["CUST_ID"].nunique()
print("Unique Brands:\n",unique_segment)

print(data[categorical].describe())

#Group Data according to  BRANDS and find descriptives:
```

```
grouped_brand= data[['CUST_ID','BRAND']].groupby(['BRAND'])
grouped_brand_cnt= grouped_brand.count()
grouped_brand_cnt=grouped_brand_cnt.sort_values(['CUST_ID'],ascending=False)
print(grouped_brand_cnt.head(10))

top_10_brand=grouped_brand_cnt.head(10)
top_10_brand=pd.DataFrame(top_10_brand)
top_10_brand.plot.bar()


#Group Data according to SEGMENTS  and find descriptives:
grouped_segment= data[['CUST_ID','SEGMENT']].groupby(['SEGMENT'])
grouped_segment_cnt= grouped_segment.count()
grouped_segment_cnt=grouped_segment_cnt.sort_values(['CUST_ID'],ascending=False)
print(grouped_segment_cnt.head(10))


top_10=grouped_segment_cnt.head(10)
top_10=pd.DataFrame(top_10)
top_10.plot.bar()

#data.head()

#dist of integer columns:
#print(data["RECENCY"])
data["RECENCY"].hist(bins="auto")
data["RECENCY"].plot(kind='box', subplots=True, layout=(3,3), sharex=False, sharey=False) #ok

np.log(data["FREQUENCY"]).hist(bins=10)

data["MONETARY"].hist()
print(data["MONETARY"])
plt.hist(np.log(data["MONETARY"]), range=(0,10000),bins='auto', color=['orange'])
plt.show()

from pandas.plotting import scatter_matrix
scatter_matrix(data[["MONETARY","FREQUENCY","RECENCY"]])
plt.show()
log_data=np.log(data[["MONETARY","FREQUENCY","RECENCY"]])
print(log_data)
pd.scatter_matrix(log_data, alpha = 0.3, figsize = (14,8), diagonal = 'kde');


#https://blog.insightdatascience.com/data-visualization-in-python-advanced-functionality-in-seaborn-
20d217f1a9a6
sns.FacetGrid(data, hue="SEGMENT", size=5) \
   .map(plt.scatter, "FREQUENCY", "MONETARY") \
   .add_legend()

sns.FacetGrid(data, hue="BRAND", size=5) \
   .map(plt.scatter, "FREQUENCY", "MONETARY") \
   .add_legend()

##correlation matrix:
names = ['FREQUENCY', 'MONETARY', 'RECENCY']
data_g = data[['FREQUENCY', 'MONETARY', 'RECENCY']]
correlations = data_g.corr()
# plot correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0,3,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names)
ax.set_yticklabels(names)
plt.show()

##########################################
#preprocessing:
#http://benalexkeen.com/feature-scaling-with-scikit-learn/
#http://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html
#1 min-max  it is sensitive to outliers.
scaler = preprocessing.MinMaxScaler()
#data=data
np_scaled = scaler.fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
df_normalized = pd.DataFrame(np_scaled)
#data['FREQUENCY']=df_normalized[0]
#data['MONETARY']=df_normalized[1]
#data['RECENCY']=df_normalized[2]
#print(df_normalized.describe())

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))
ax1.set_title('Before  Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
```

```
ax2.set_title('After  Scaler')
sns.kdeplot(df_normalized[0], ax=ax2)
sns.kdeplot(df_normalized[1], ax=ax2)
sns.kdeplot(df_normalized[2], ax=ax2)
plt.show()

#2 standard -data not normal dist.

scaler = preprocessing.StandardScaler() #MaxAbsScaler
np_scaled = scaler.fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
df_normalized = pd.DataFrame(np_scaled)
#print(df_normalized.describe())
#plt.hist(df_normalized)

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))
ax1.set_title('Before  Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
ax2.set_title('After  Scaler')
sns.kdeplot(df_normalized[0], ax=ax2)
sns.kdeplot(df_normalized[1], ax=ax2)
sns.kdeplot(df_normalized[2], ax=ax2)
plt.show()

#3 rubost: best for outliers!

scaler = preprocessing.RobustScaler(quantile_range=(25, 75))
np_scaled = scaler.fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
df_normalized = pd.DataFrame(np_scaled)
print(df_normalized.describe())

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))

ax1.set_title('Before Rubost Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
ax2.set_title('After  Scaler')
sns.kdeplot(df_normalized[0], ax=ax2)
sns.kdeplot(df_normalized[1], ax=ax2)
sns.kdeplot(df_normalized[2], ax=ax2)
plt.show()

#Standardization refers to shifting the distribution of each attribute to have a mean of zero and a
standard deviation of one (unit variance).
#standardized_X = preprocessing.scale(data[['FREQUENCY', 'MONETARY', 'RECENCY']]) StandardScaler ile aynı
yani çöp.
#o'Data after sample-wise L2 normalizing'  Normalizer().fit_transform(X)
normalized = Normalizer().fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
s_data=pd.DataFrame(normalized)
print(s_data.describe())
#plt.hist(standardized_X)

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))
ax1.set_title('Before  Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
ax2.set_title('After  Scaler')
sns.kdeplot(s_data[0], ax=ax2)
sns.kdeplot(s_data[1], ax=ax2)
sns.kdeplot(s_data[2], ax=ax2)
plt.show()

names = ['FREQUENCY', 'MONETARY', 'RECENCY']
data_g = s_data #data[['FREQUENCY', 'MONETARY', 'RECENCY']]
correlations = data_g.corr()
# plot correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0,3,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names)
ax.set_yticklabels(names)
plt.show()


#Quantile: the best!
scaler = QuantileTransformer(output_distribution='normal')
    #preprocessing.StandardScaler()  #QuantileTransformer(output_distribution='uniform')
np_scaled = scaler.fit_transform(data[['FREQUENCY', 'MONETARY', 'RECENCY']])
df_normalized = pd.DataFrame(np_scaled)
print(df_normalized.describe())
#plt.hist(df_normalized)
```

```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(6, 5))
ax1.set_title('Before  Scaling')
sns.kdeplot(data['FREQUENCY'], ax=ax1)
sns.kdeplot(data['MONETARY'], ax=ax1)
sns.kdeplot(data['RECENCY'], ax=ax1)
ax2.set_title('After  Scaler')
sns.kdeplot(df_normalized[0], ax=ax2)
sns.kdeplot(df_normalized[1], ax=ax2)
sns.kdeplot(df_normalized[2], ax=ax2)
plt.show()

##correlation matrix after scaling:

names = ['FREQUENCY', 'MONETARY', 'RECENCY']
data_g = df_normalized #data[['FREQUENCY', 'MONETARY', 'RECENCY']]
correlations = data_g.corr()
# plot correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0,3,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names)
ax.set_yticklabels(names)
plt.show()

##cleaning after transformation(scaling):
#rename transformed columns:
df_normalized.rename(columns={0:"FREQUENCY_SCALED",1:"MONETARY_SCALED",2:"RECENCY_SCALED"},inplace=True)
print(df_normalized.info())


#merge scaled data nad other categorical columns:
q_transformed_data= pd.concat([df_normalized,data],axis=1)
print(q_transformed_data.info())

#need to drop unnecessary columns:
#del q_transformed_data["CUST_ID"]
del q_transformed_data["MONETARY"]
del q_transformed_data["FREQUENCY"]
del q_transformed_data["RECENCY"]

print(q_transformed_data.info())
print(q_transformed_data.describe())

#create dummy from segment:
lb_style = LabelBinarizer()
lb_results = lb_style.fit_transform(q_transformed_data["SEGMENT"])
lb_results=pd.DataFrame(lb_results, columns=lb_style.classes_)
#print(pd.DataFrame(lb_results, columns=lb_style.classes_).head())

q_transformed_data= pd.concat([q_transformed_data,lb_results],axis=1)
#print(q_transformed_data.info())
print(q_transformed_data.shape)

del q_transformed_data["SEGMENT"]
del q_transformed_data["BRAND"]
del q_transformed_data["CUST_ID"]

print(data.loc[0])
print(q_transformed_data.loc[0]["TELEFON"])


#COR. MATRIX AFTER DUMMIES:

names = q_transformed_data.columns #['FREQUENCY', 'MONETARY', 'RECENCY']
data_g = q_transformed_data #data[['FREQUENCY', 'MONETARY', 'RECENCY']]
correlations = data_g.corr()
# plot correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0,41,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names,rotation='vertical')
ax.set_yticklabels(names) # fontdict={'verticalalignment': 'baseline'})
plt.show()

##################################MODELING#############################
##modeling
from time import time  # to keep track of the processing time
from sklearn import metrics
from sklearn.cluster import KMeans
```

```
print(q_transformed_data.info())  #11117   , 13897

###############k-means

#elbow curve:
Nc = range(1, 10)
kmeans = [KMeans(n_clusters=i) for i in Nc]
kmeans
score = [kmeans[i].fit(q_transformed_data).score(q_transformed_data) for i in range(len(kmeans))]
print(score)
pl.plot(Nc,score)
pl.xlabel('Number of Clusters')
pl.ylabel('Score')
pl.title('Elbow Curve')
pl.show()


def k_means(q_transformed_data):
    num_clusters = [4] #[3,4,5,6,7,8]
    sil_dic={}
    sil_ecd_dic={}
    for i in num_clusters:
        print("************************ n_cluster= %d ************************" %i)
        kmeans = KMeans(init='k-means++',n_clusters= i, n_jobs= -1)
        t0 = time()
        kmeans_output=kmeans.fit(q_transformed_data)
        t1 = time() - t0
        print("K-means took: %.3f ", t1)

        df = pd.DataFrame(kmeans.cluster_centers_)
        df['count'] = pd.Series(kmeans.labels_).value_counts()  # Number of each clusters
        print("Number of elements for each cluster: \t")
        print(df['count'])
        #print("K-means took:\t%.3f "% t1, "sec")
        silhouette_cosine= metrics.silhouette_score(q_transformed_data,kmeans.labels_, metric='cosine',
sample_size=q_transformed_data.shape[0])
        silhouette_ecludiene= metrics.silhouette_score(q_transformed_data,kmeans.labels_,
metric='euclidean', sample_size=q_transformed_data.shape[0])
        ch_score=metrics.calinski_harabaz_score(q_transformed_data,kmeans_output.labels_)
        print("silhouette score %.3f" %ch_score)
        print("silhouette score %.3f" %silhouette_ecludiene)
        sil_dic[i]=silhouette_cosine
        sil_ecd_dic[i]=silhouette_ecludiene

    print(sil_dic) # kesin4 mantıklısı gibi.
    #print(sil_ecd_dic)
    return sil_dic,sil_ecd_dic

sil_dic,sil_ecd_dic = k_means(q_transformed_data)

###################PCA
#PCA ile daha iyi! 4 cluster ile PCA li.
from sklearn.decomposition import PCA
pca = PCA(n_components=3)
reduced_data = pca.fit_transform(q_transformed_data)
#reduced_data = PCA(n_components=2).fit_transform(q_transformed_data)
print(pca.explained_variance_ratio_)

print(reduced_data.shape)
Nc = range(1, 10)
kmeans = [KMeans(n_clusters=i) for i in Nc]
kmeans
score = [kmeans[i].fit(reduced_data).score(reduced_data) for i in range(len(kmeans))]
print(score)
pl.plot(Nc,score)
pl.xlabel('Number of Clusters')
pl.ylabel('Score')
pl.title('Elbow Curve')
pl.show()

#K-MEANS after PCA (K-Means on dimensionaly reduced data):
kmeans = KMeans(init='k-means++',n_clusters= 4, n_jobs= -1)
t0 = time()
kmeansoutput=kmeans.fit(reduced_data)
t1 = time() - t0
print("K-Means on reduced data:",t1)
df = pd.DataFrame(kmeans.cluster_centers_)
df['count'] = pd.Series(kmeans.labels_).value_counts()  # Number of each clusters
print("Number of elements for each cluster: \t")
print(df['count'])

silhouette_cosine= metrics.silhouette_score(reduced_data,kmeans.labels_, metric='cosine',
sample_size=reduced_data.shape[0])
silhouette_ecludiene= metrics.silhouette_score(reduced_data,kmeans.labels_, metric='euclidean',
sample_size=reduced_data.shape[0])
print("calinski_harabaz_Score:",metrics.calinski_harabaz_score(reduced_data,kmeansoutput.labels_))
```

```
print(silhouette_ecludiene)
#0.4241755199146903

#print(reduced_data.shape)
#plot clusters of K-Means on reduced data:
pl.figure('4 Cluster K-Means')
pl.scatter(reduced_data[:, 0],reduced_data[:, 1], c=kmeansoutput.labels_)
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)
pl.xlabel('Principle Component 1')
pl.ylabel('Principle Component 2')
pl.title('4 Cluster K-Means')
pl.show()

#################################  mean-shift clustering
from sklearn.cluster import MeanShift, estimate_bandwidth

# Compute clustering with MeanShift
# The following bandwidth can be automatically detected using
X=reduced_data
bandwidth = estimate_bandwidth(reduced_data, quantile=0.1, n_samples=50)

ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
t0 = time()
meanshift_output=ms.fit(X)
t1 = time() - t0
labels = ms.labels_
cluster_centers = ms.cluster_centers_
df = pd.DataFrame(cluster_centers)
df['count'] = pd.Series(labels).value_counts()
print("Number of element due to Clusters:\n",df['count'])
print("Runtime for Mean-Shift: ",t1)
print("Silhouette Coefficient: %0.3f" % metrics.silhouette_score(X, labels))
print("CH Coefficient: %0.3f" % metrics.calinski_harabaz_score(reduced_data,labels))

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)
print("number of estimated clusters : %d" % n_clusters_)

# Plot clusters of Mean-Shift on reduced data:
import matplotlib.pyplot as plt
from itertools import cycle

plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()



######## AgglomerativeClustering
from sklearn.cluster import AgglomerativeClustering

X=reduced_data

clt = AgglomerativeClustering(linkage='average',
                              affinity='euclidean',
                              n_clusters=4)
t0 = time()
model = clt.fit(X)
t1 = time() - t0
labels = clt.labels_
print(labels)
df = pd.DataFrame(reduced_data,labels)
df['count'] = pd.Series(labels).value_counts()
print("Runtime for Agglomerative: ",t1)
print("Silhouette Coefficient: %0.3f" % metrics.silhouette_score(X, labels))
print("CH Coefficient: %0.3f" % metrics.calinski_harabaz_score(reduced_data,labels))

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)

# Plot clusters of AC on reduced data:
import matplotlib.pyplot as plt
from itertools import cycle
```

```
plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()

##done.
```