University of Missouri-Kansas City

School of Computing and Engineering

ICP:	ICP-06
Course Name:	Big Data Analytics and Applications
Course ID:	COMP-SCI 5542
Semester /Session :	Spring 2022

Student ID :	**Student Name**
16334245	Mustavi Islam
16321217	Keenan Flynn

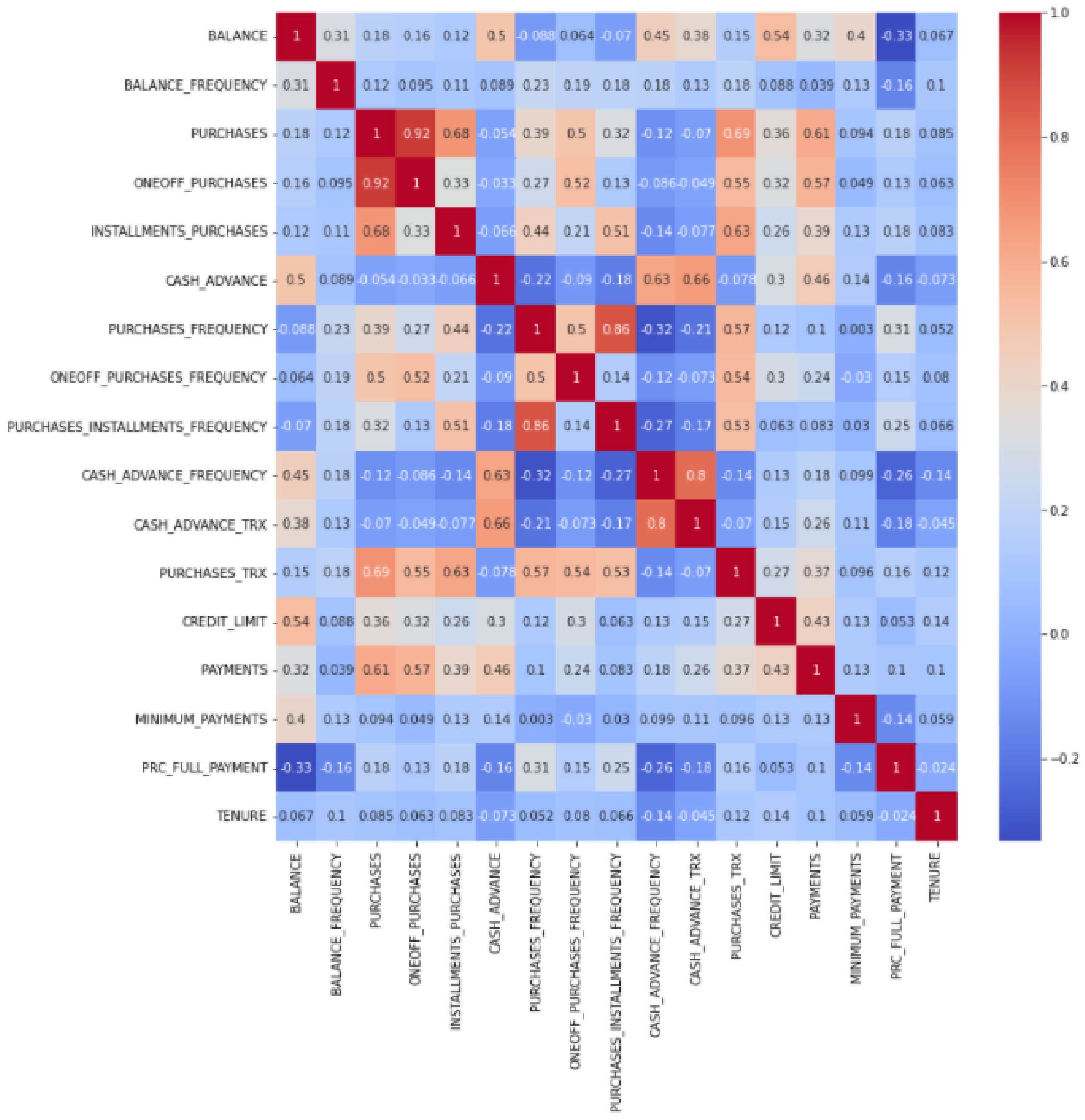
Description of the Problem :

Unsupervised Learning help us solve problems with unlabeled data. A majority of data is unstructured and lacks labels. Unsupervised Learning can be used to group this kind of data into clusters. One most well known algorithm in unsupervised learning is KMeans clustering. This algorithm partitions data into groups based on similarity between data points. Similarity is found by calculating the Euclidean distance between these data points.

In this ICP, we have used a dataset with Credit Card usage among various users. Our goal is to segment these users into buckets so that we can target specific groups. For example, if we wanted to target a group of high spenders with an ad campaign, we would first need to realize who that high spending group consists of. We can't manually assign labels to all of the high spenders as that would take too much time. Instead we used unsupervised learning and clustering.

Description of Solution :

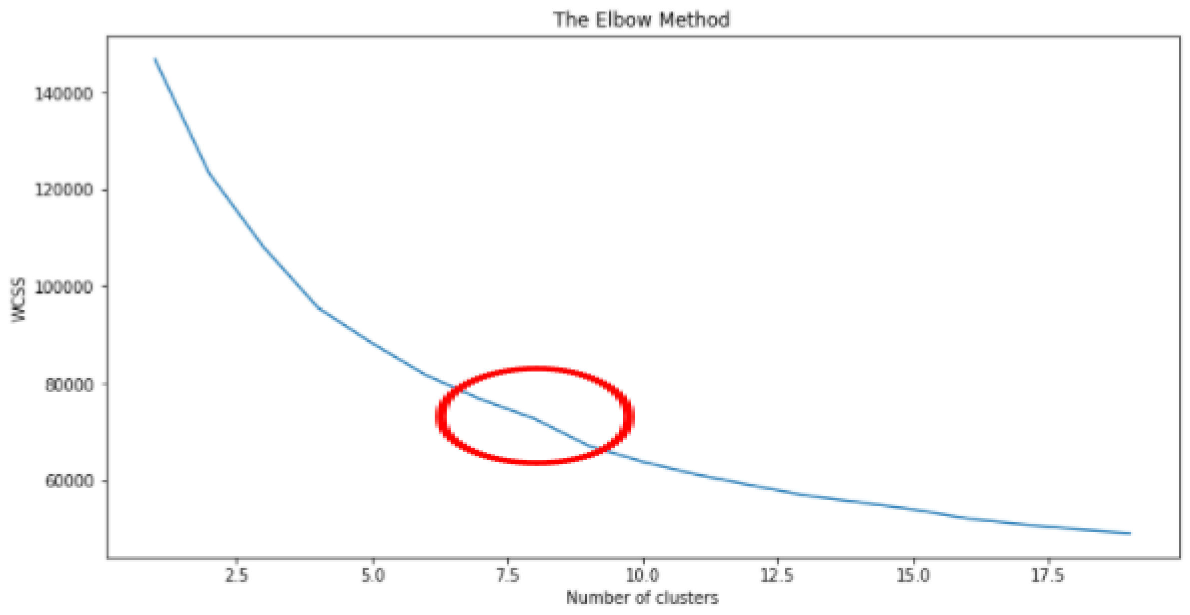
1. Get the dataset
    a. The credit card dataset is a commonly used dataset. We downloaded it from Kaggle but you can also download it from the UCI repository.
2. Data Exploration
    a. The traditional KMeans algorithm can only be used with numeric data so let's explore our dataset.
    b. We can use df.info() to get the datatypes of each column. Every column is numeric except for the CustID column (type Object). Customer ID doesn't tell us anything useful about the customers spending so we drop that column.
    c. The dataset also has about 300 rows that contain null values. Unlike Neural Networks, KMeans does not need a lot of data. We can drop these null rows.
    d. We can use df.describe() to get statistics about the data. KMeans is sensitive to outliers so we pay close attention to the standard deviation. The standard deviation ranges from 0.2 to 3700. Before we use KMeans we will want to scale our data.
    e. Let's get an understanding on the correlation in our data. We can do this by creating a heatmap using Seaborn. From this heatmap we can see that correlation exists in the data.



3. Pre-Processing
    a. We will pre-process the data by scaling it. The StandardScalar() method from Sci-Kit Learn scales the data so that it has a mean of 0 and a standard deviation of 1.
    b. We can see this by using df.describe() again.
    c. Our data has the shape `(8636, 17)`, we will keep this in mind for the visualization step.
4. KMeans
    a. To determine the optimal number of clusters for our dataset, we will use the elbow method. The elbow method plots # clusters against the inertia.

b. Ideally, we want minimal inertia but we also want to deal with a small number of clusters. We choose the number of clusters by visualizing the inertia decline and deciding where the slope starts to flatten out.

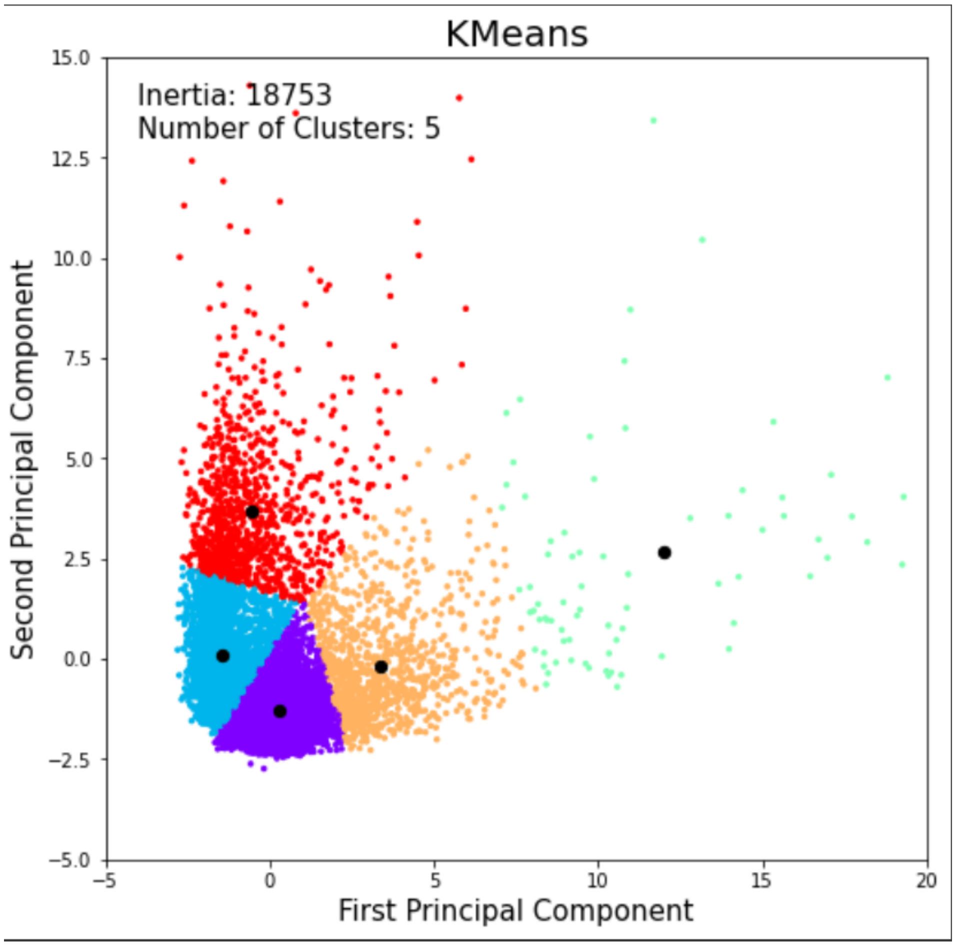
c. To create this plot, we looped through 20 times and created/fit 20 different KMeans objects and measured their inertia. Here were our results.

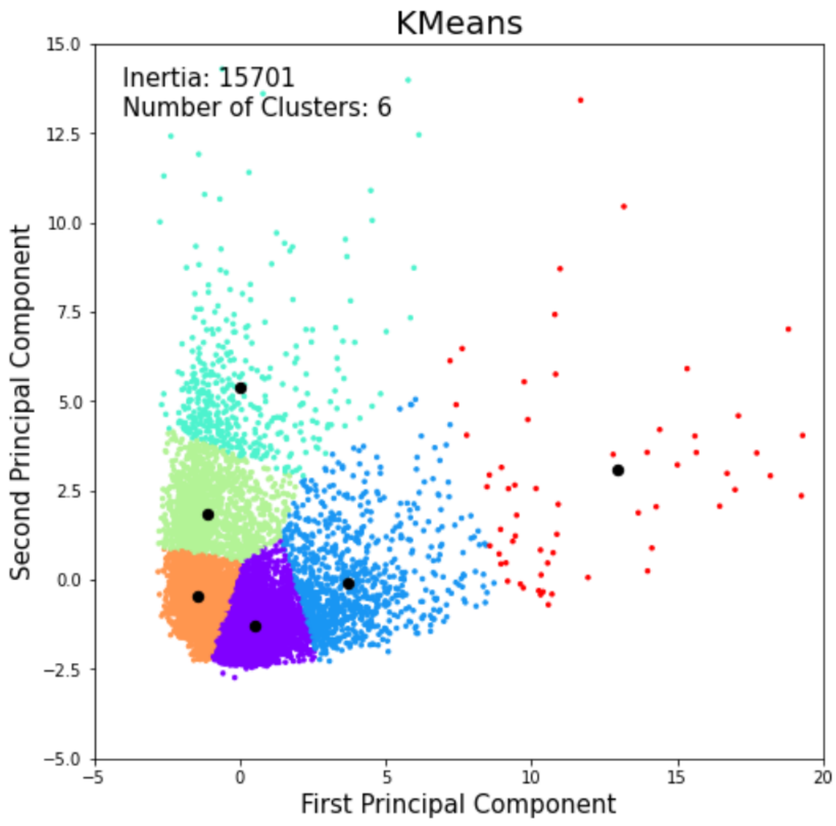


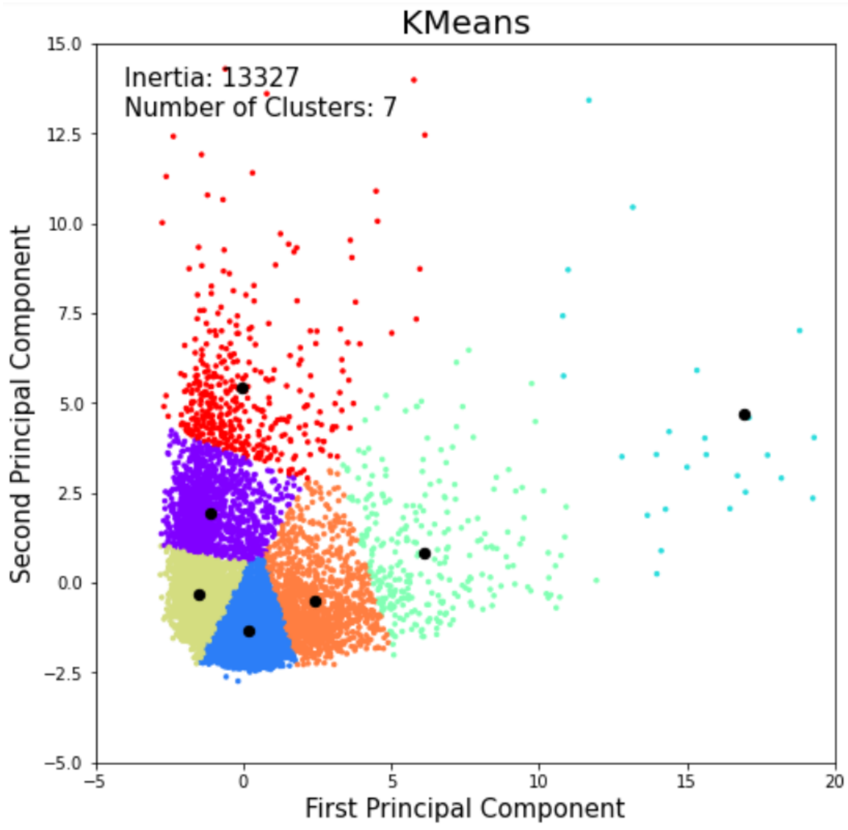
d. We can now recreate KMeans objects with 5 to 9 clusters. These objects can be used to predict the labels of our dataset as well as any new data that we have. We use the KMeans.fit() and KMeans.predict() methods.

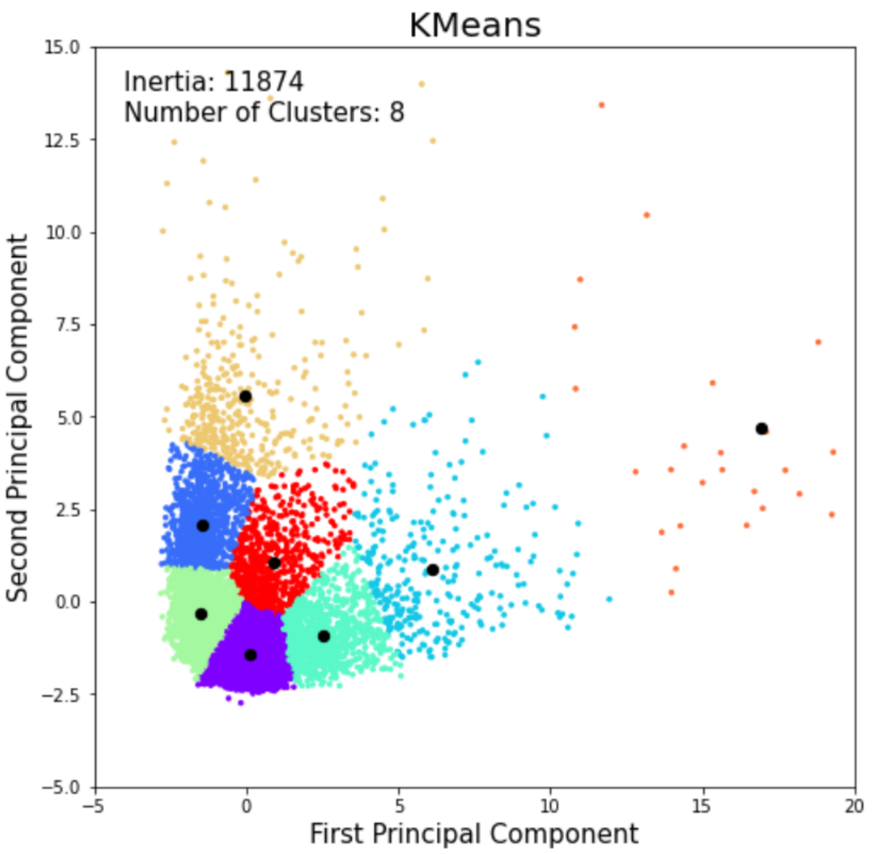
5. Visualization
    a. Our dataset contains 17 dimensions as stated earlier. This is too many dimensions for humans to imagine. We need to get our data into a lower dimensional space.
    b. We can do this by using principal component analysis. This process creates a new dataset with columns that are calculated by mixing all of the old columns together. The PCA process does this while maximizing the variance of the dataset so that it can try to capture the characteristics of the original dataset. In this way, we can decompose our 17 dimensions into 2 dimensions without losing too much of the data.
    c. With the new PCA data, we ran KMeans algorithm again with 5 to 9 clusters. We then plotted the first 2 principal components along with the respective cluster label for each data point.
    d. The PCA KMeans will not perfectly reproduce the clusters from the original KMeans, but it is good enough for visualization purposes. Here is the resultant graph. Each cluster is grouped around a black dot which is the centroid of that group.

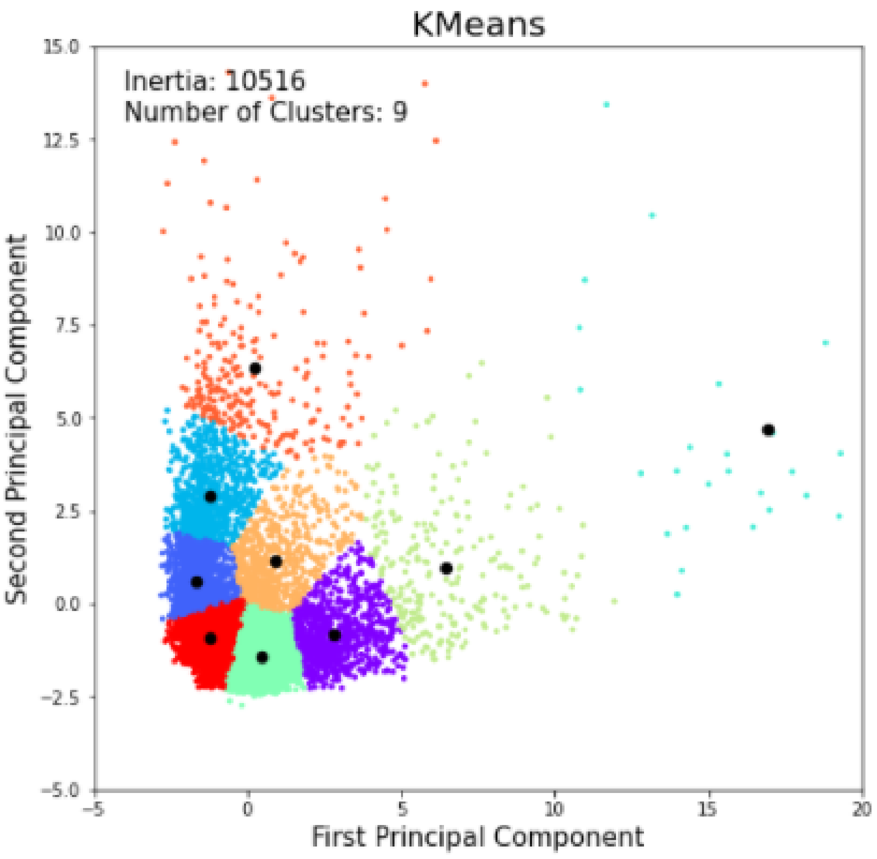
e. The optimal number of clusters is a subjective measure. Having too many clusters can make subsequent learning algorithms less efficient. For this exercise, we have decided that 9 clusters are the optimal amount of clusters because this has the lowest inertia.




KMeans
Inertia: 15701
Number of Clusters: 6
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−5
0
5
10
15
20
Second Principal Component
First Principal Component


KMeans
Inertia: 13327
Number of Clusters: 7
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−5
0
5
10
15
20
Second Principal Component
First Principal Component


KMeans
Inertia: 11874
Number of Clusters: 8
Second Principal Component
First Principal Component
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−5
0
5
10
15
20


KMeans
Inertia: 10516
Number of Clusters: 9
Second Principal Component
First Principal Component
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−5
0
5
10
15
20

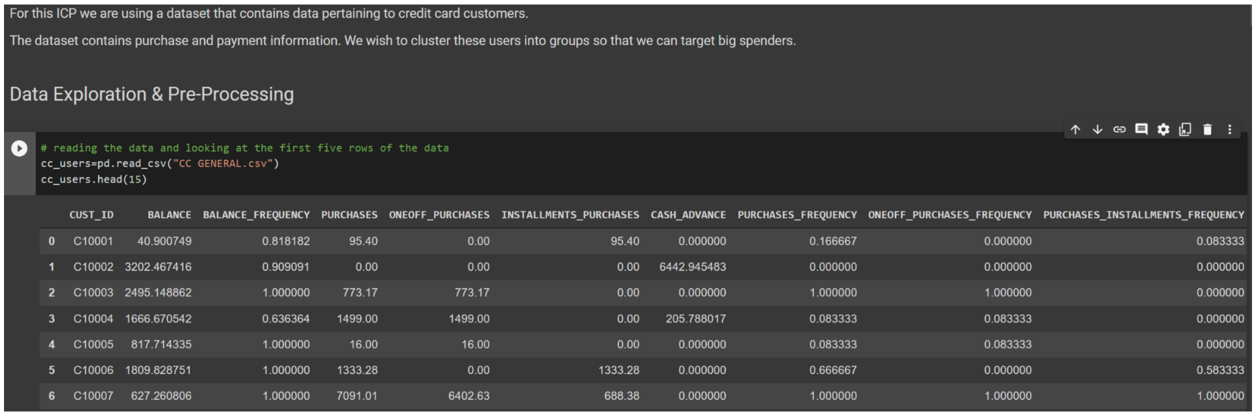
Challenges:
The biggest challenge of this ICP was finding a good dataset. Many of the datasets that we found online through Kaggle and UCI had already been pre-processed and labeled for supervised learning. This dataset was chosen because of the idea that we could group users together by their spending habits.

Another challenge we had was graphing the clusters. Our dataset had 17 columns or independent variables. It is impossible to graph this many dimensions. This is where PCA helped. The drawback of PCA is that we lose some of the data when we decompose our dataset. It will be important to keep in mind that a PCA KMeans model allows us to visualize these clusters. However if we have new data that we want to run through the model, we will not want to use the PCA model because some data is lost.

Learning Outcomes :
1. We learned when to apply unsupervised vs supervised learning.
2. We learned how to apply the Elbow method on a dataset.
3. We learned that you can use the KMeans++ initializer to get a better beginning iteration.
4. We learned that you can use PCA to graph a high-dimensional dataset

Resources:
https://www.kaggle.com/arjunbhasin2013/ccdata?select=CC+GENERAL.csv
Kaggle data set

Video Link: https://youtu.be/PYujyuT0HDA

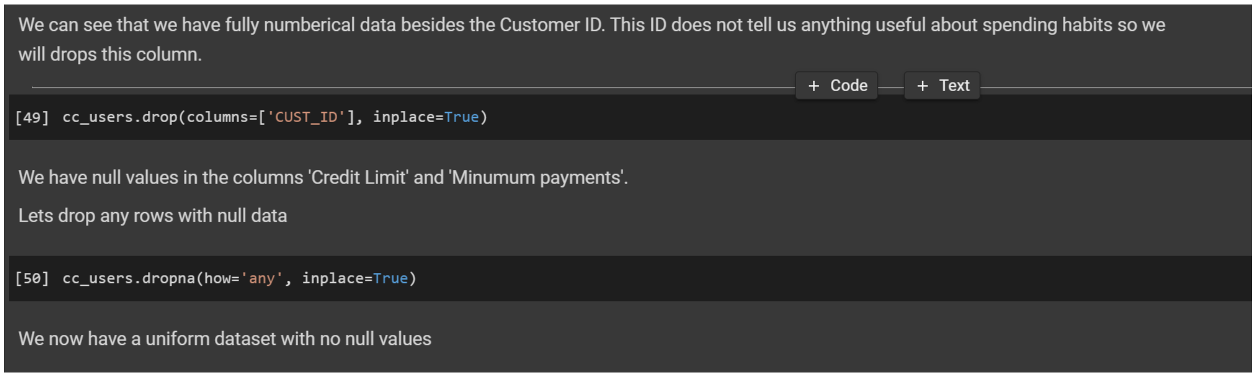
Screenshots :

For this ICP we are using a dataset that contains data pertaining to credit card customers.

The dataset contains purchase and payment information. We wish to cluster these users into groups so that we can target big spenders.

Data Exploration & Pre-Processing

```
# reading the data and looking at the first five rows of the data
cc_users=pd.read_csv("CC GENERAL.csv")
cc_users.head(15)
```

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY
0	C10001	40.900749	0.818182	95.40	0.00	95.40	0.000000	0.166667	0.000000	0.083333
1	C10002	3202.467416	0.909091	0.00	0.00	0.00	6442.945483	0.000000	0.000000	0.000000
2	C10003	2495.148862	1.000000	773.17	773.17	0.00	0.000000	1.000000	1.000000	0.000000
3	C10004	1666.670542	0.636364	1499.00	1499.00	0.00	205.788017	0.083333	0.083333	0.000000
4	C10005	817.714335	1.000000	16.00	16.00	0.00	0.000000	0.083333	0.083333	0.000000
5	C10006	1809.828751	1.000000	1333.28	0.00	1333.28	0.000000	0.666667	0.000000	0.583333
6	C10007	627.260806	1.000000	7091.01	6402.63	688.38	0.000000	1.000000	1.000000	1.000000


We can see that we have fully numberical data besides the Customer ID. This ID does not tell us anything useful about spending habits so we will drops this column.

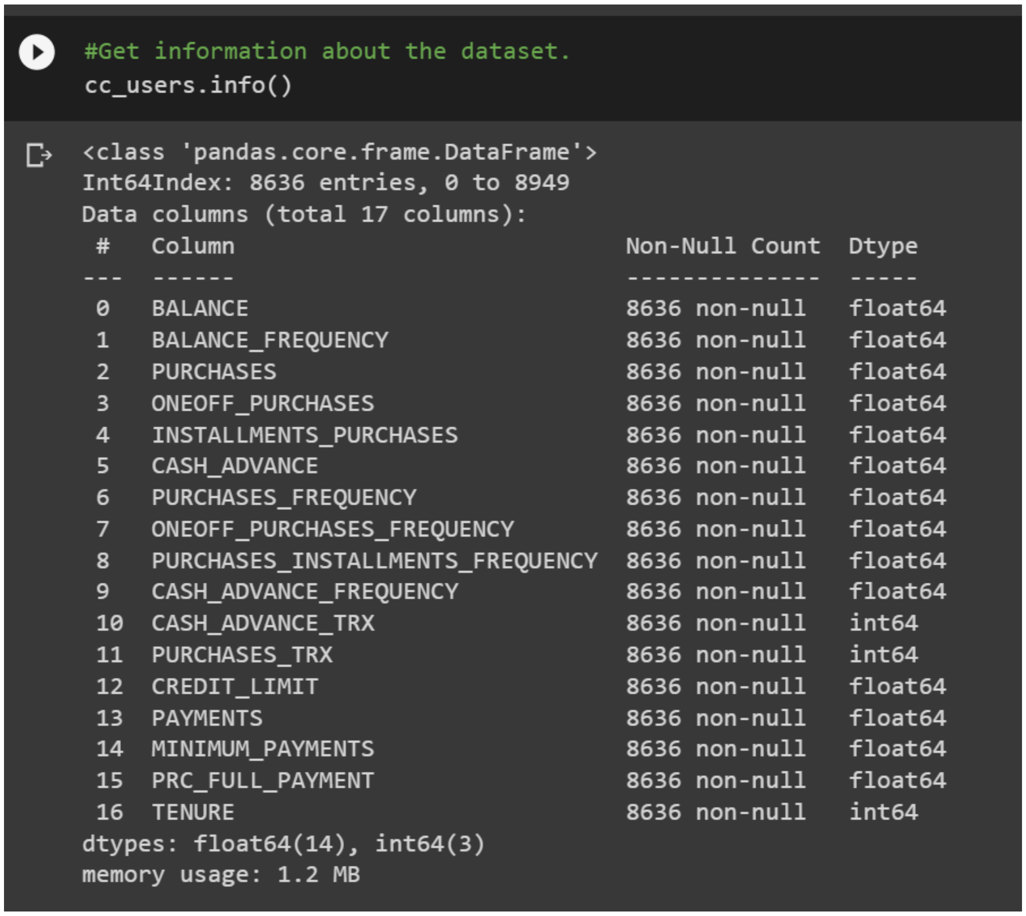
```
[49] cc_users.drop(columns=['CUST_ID'], inplace=True)
```

We have null values in the columns 'Credit Limit' and 'Minumum payments'.

Lets drop any rows with null data

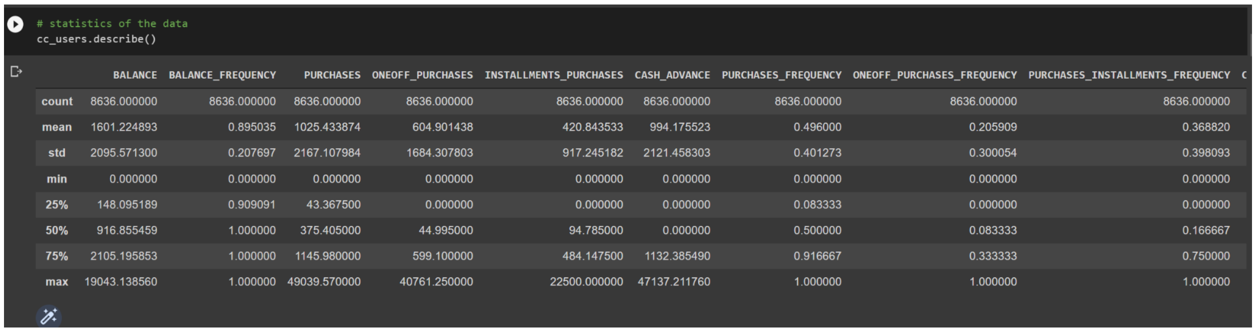
```
[50] cc_users.dropna(how='any', inplace=True)
```

We now have a uniform dataset with no null values



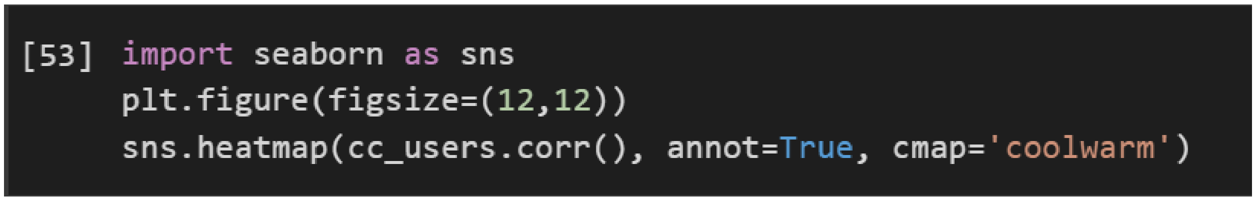
```
#Get information about the dataset.
cc_users.info()
```

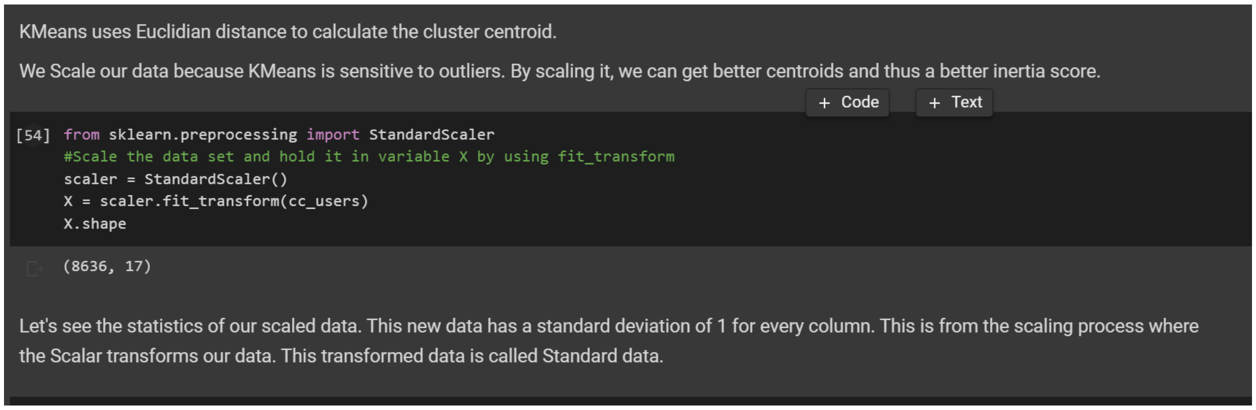
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8636 entries, 0 to 8949
Data columns (total 17 columns):
 #   Column                            Non-Null Count  Dtype  
---  ------                            --------------  -----  
 0   BALANCE                           8636 non-null   float64
 1   BALANCE_FREQUENCY                 8636 non-null   float64
 2   PURCHASES                         8636 non-null   float64
 3   ONEOFF_PURCHASES                  8636 non-null   float64
 4   INSTALLMENTS_PURCHASES            8636 non-null   float64
 5   CASH_ADVANCE                      8636 non-null   float64
 6   PURCHASES_FREQUENCY               8636 non-null   float64
 7   ONEOFF_PURCHASES_FREQUENCY        8636 non-null   float64
 8   PURCHASES_INSTALLMENTS_FREQUENCY  8636 non-null   float64
 9   CASH_ADVANCE_FREQUENCY            8636 non-null   float64
 10  CASH_ADVANCE_TRX                  8636 non-null   int64  
 11  PURCHASES_TRX                     8636 non-null   int64  
 12  CREDIT_LIMIT                      8636 non-null   float64
 13  PAYMENTS                          8636 non-null   float64
 14  MINIMUM_PAYMENTS                  8636 non-null   float64
 15  PRC_FULL_PAYMENT                  8636 non-null   float64
 16  TENURE                            8636 non-null   int64  
dtypes: float64(14), int64(3)
memory usage: 1.2 MB
```



```
# statistics of the data
cc_users.describe()
```

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY	C
count	8636.000000	8636.000000	8636.000000	8636.000000	8636.000000	8636.000000	8636.000000	8636.000000	8636.000000	
mean	1601.224893	0.895035	1025.433874	604.901438	420.843533	994.175523	0.496000	0.205909	0.368820	
std	2095.571300	0.207697	2167.107984	1684.307803	917.245182	2121.458303	0.401273	0.300054	0.398093	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	148.095189	0.909091	43.367500	0.000000	0.000000	0.000000	0.083333	0.000000	0.000000	
50%	916.855459	1.000000	375.405000	44.995000	94.785000	0.000000	0.500000	0.083333	0.166667	
75%	2105.195853	1.000000	1145.980000	599.100000	484.147500	1132.385490	0.916667	0.333333	0.750000	
max	19043.138560	1.000000	49039.570000	40761.250000	22500.000000	47137.211760	1.000000	1.000000	1.000000	


```
[53] import seaborn as sns
     plt.figure(figsize=(12,12))
     sns.heatmap(cc_users.corr(), annot=True, cmap='coolwarm')
```


KMeans uses Euclidian distance to calculate the cluster centroid.

We Scale our data because KMeans is sensitive to outliers. By scaling it, we can get better centroids and thus a better inertia score.

```
[54] from sklearn.preprocessing import StandardScaler
     #Scale the data set and hold it in variable X by using fit_transform
     scaler = StandardScaler()
     X = scaler.fit_transform(cc_users)
     X.shape
```

```
(8636, 17)
```

Let's see the statistics of our scaled data. This new data has a standard deviation of 1 for every column. This is from the scaling process where the Scalar transforms our data. This transformed data is called Standard data.

KMeans

Let's do KMeans clustering on our scaled data. We can iterate through different numbers of clusters and get an inertia score.

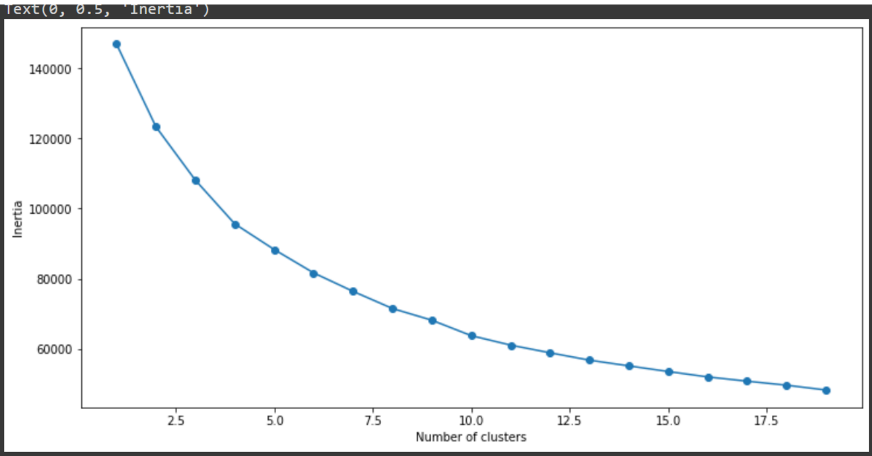
Our goal is to have meaningful clusters that characterize all of the subgroups in the data.

We want to avoid choosing too many clusters because this can lead to bloat. It is best to take the smallest amount of clusters that also has a good inertia score.

We can do this by using the elbow method.

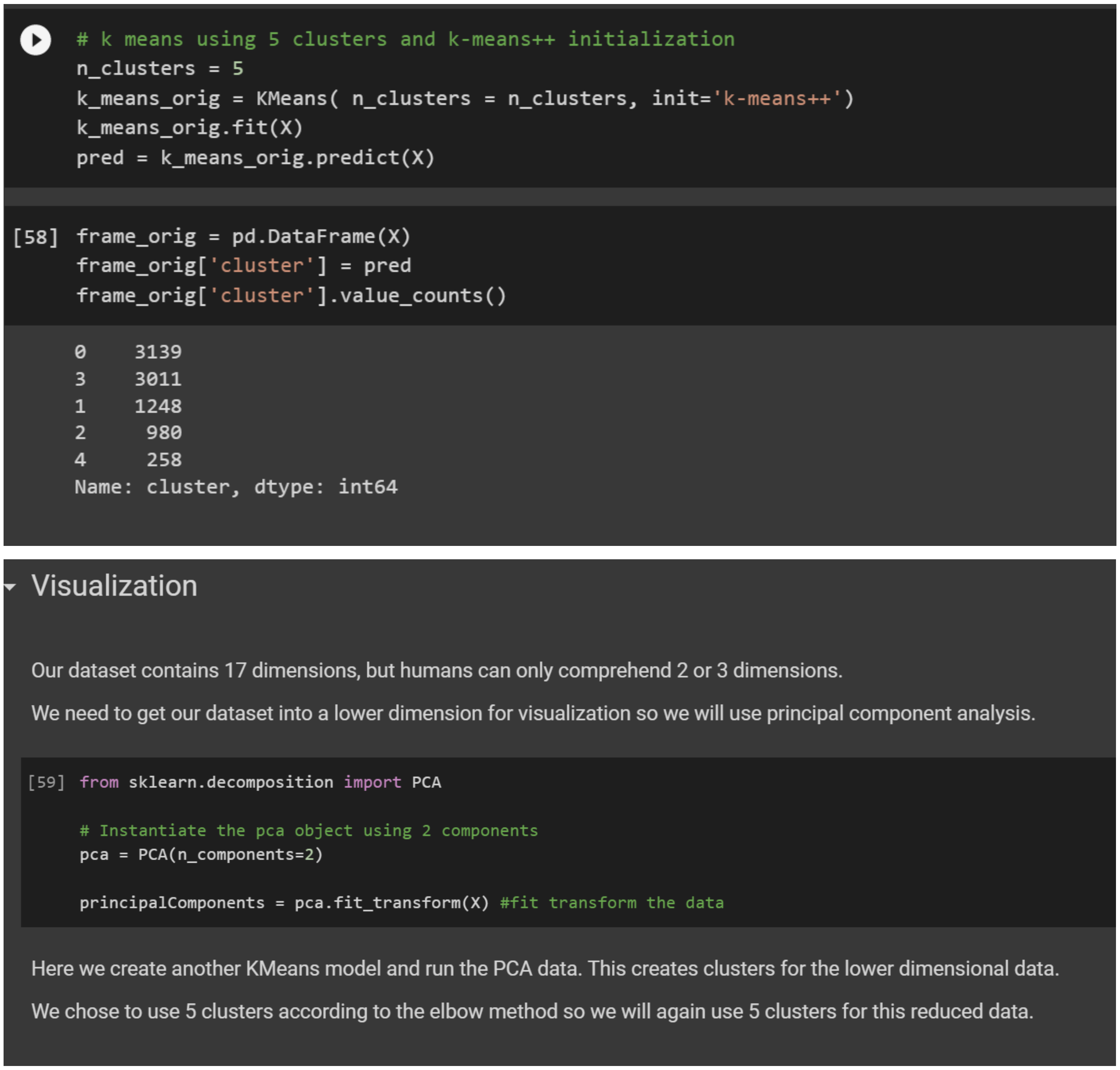
```
[56] # fitting multiple k-means algorithms and storing the values in an empty list
     SSE = []
     for cluster in range(1,20):
         kmeans = KMeans(n_clusters = cluster, init='k-means++')
         kmeans.fit(X)
         SSE.append(kmeans.inertia_)

     # converting the results into a dataframe and plotting them
     frame = pd.DataFrame({'Cluster':range(1,20), 'SSE':SSE})
     plt.figure(figsize=(12,6))
     plt.plot(frame['Cluster'], frame['SSE'], marker='o')
     plt.xlabel('Number of clusters')
     plt.ylabel('Inertia')
```

```
Text(0, 0.5, 'Inertia')
```



From the elbow plot, we choose k=5 to k=9 clusters for the data. We then create this new KMeans model and predict labels with a new dataset.

From the elbow plot, we choose 9 clusters for the data. We then create this new KMeans model and predict labels with a new dataset.



```
# k means using 5 clusters and k-means++ initialization
n_clusters = 5
k_means_orig = KMeans( n_clusters = n_clusters, init='k-means++')
k_means_orig.fit(X)
pred = k_means_orig.predict(X)
```

```
[58] frame_orig = pd.DataFrame(X)
     frame_orig['cluster'] = pred
     frame_orig['cluster'].value_counts()
```

```
0    3139
3    3011
1    1248
2     980
4     258
Name: cluster, dtype: int64
```

Visualization

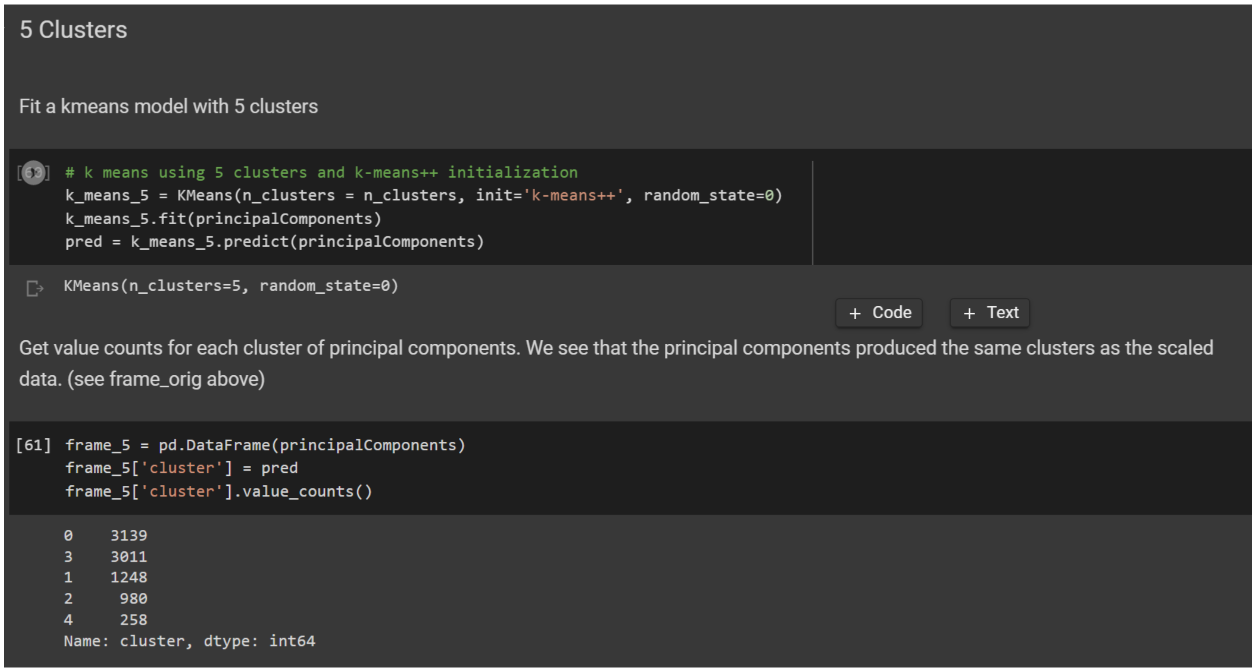
Our dataset contains 17 dimensions, but humans can only comprehend 2 or 3 dimensions.

We need to get our dataset into a lower dimension for visualization so we will use principal component analysis.

```
[59] from sklearn.decomposition import PCA

     # Instantiate the pca object using 2 components
     pca = PCA(n_components=2)

     principalComponents = pca.fit_transform(X) #fit transform the data
```

Here we create another KMeans model and run the PCA data. This creates clusters for the lower dimensional data.

We chose to use 5 clusters according to the elbow method so we will again use 5 clusters for this reduced data.



5 Clusters

Fit a kmeans model with 5 clusters

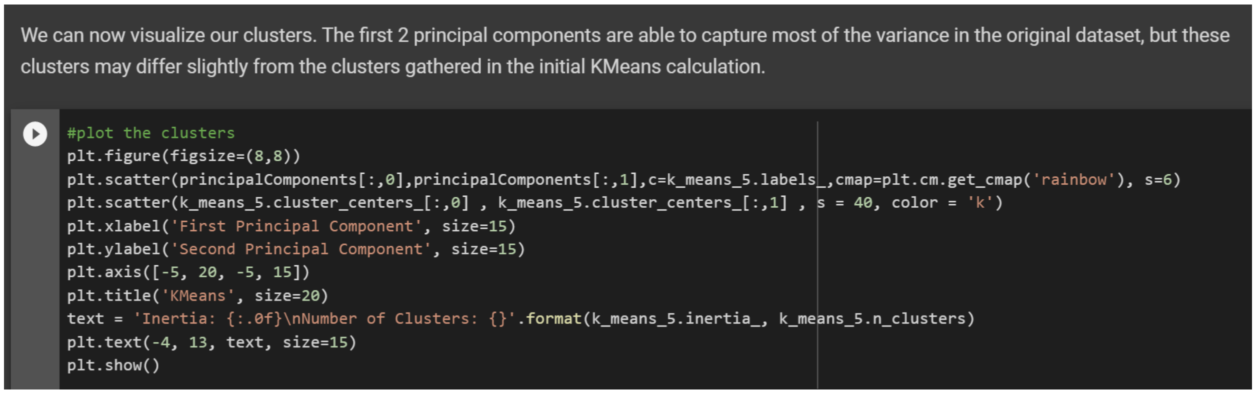
```
[ ] # k means using 5 clusters and k-means++ initialization
    k_means_5 = KMeans(n_clusters = n_clusters, init='k-means++', random_state=0)
    k_means_5.fit(principalComponents)
    pred = k_means_5.predict(principalComponents)
```

```
KMeans(n_clusters=5, random_state=0)
```

Get value counts for each cluster of principal components. We see that the principal components produced the same clusters as the scaled data. (see frame_orig above)

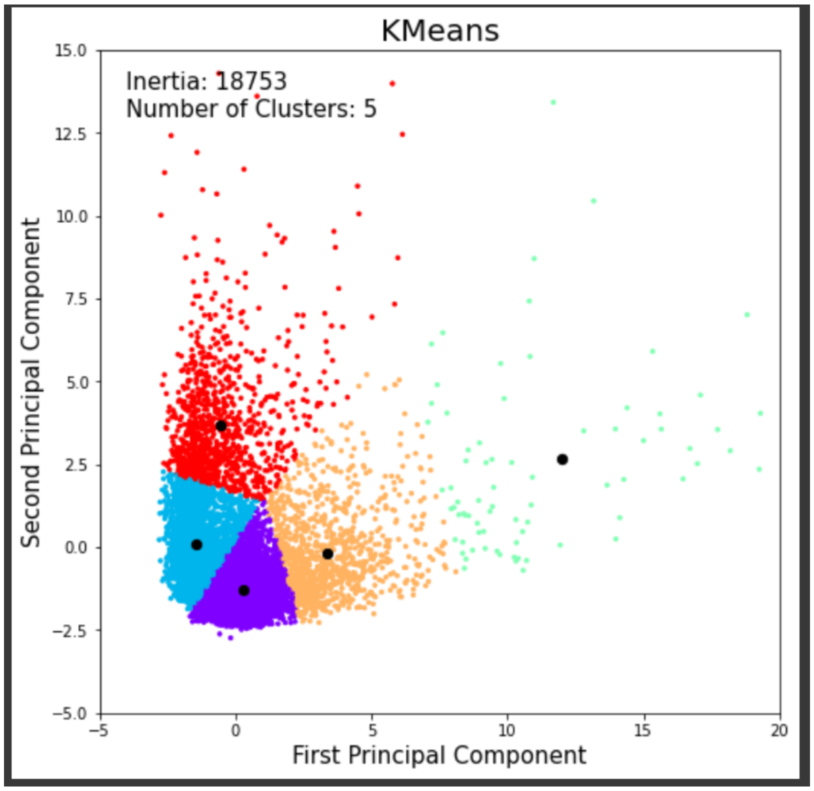
```
[61] frame_5 = pd.DataFrame(principalComponents)
     frame_5['cluster'] = pred
     frame_5['cluster'].value_counts()
```

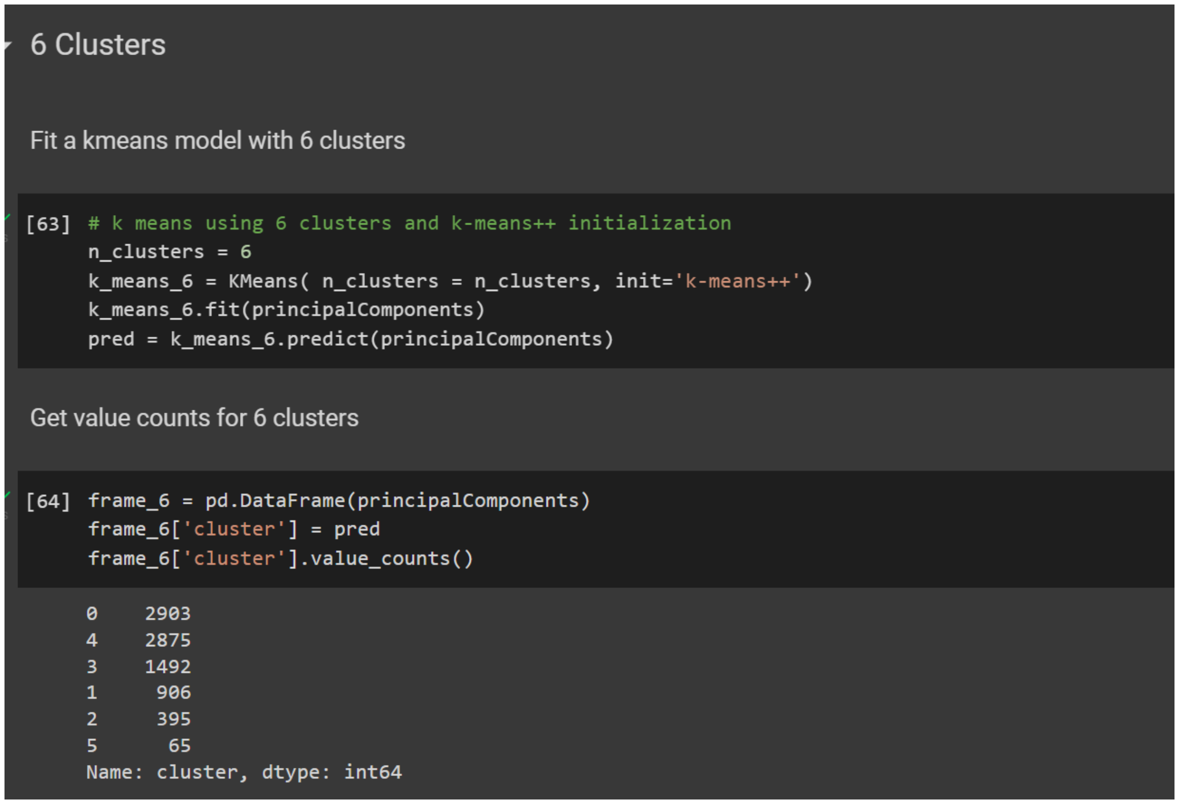
```
0    3139
3    3011
1    1248
2     980
4     258
Name: cluster, dtype: int64
```



We can now visualize our clusters. The first 2 principal components are able to capture most of the variance in the original dataset, but these clusters may differ slightly from the clusters gathered in the initial KMeans calculation.

```
#plot the clusters
plt.figure(figsize=(8,8))
plt.scatter(principalComponents[:,0],principalComponents[:,1],c=k_means_5.labels_,cmap=plt.cm.get_cmap('rainbow'), s=6)
plt.scatter(k_means_5.cluster_centers_[:,0] , k_means_5.cluster_centers_[:,1] , s = 40, color = 'k')
plt.xlabel('First Principal Component', size=15)
plt.ylabel('Second Principal Component', size=15)
plt.axis([-5, 20, -5, 15])
plt.title('KMeans', size=20)
text = 'Inertia: {:.0f}\nNumber of Clusters: {}'.format(k_means_5.inertia_, k_means_5.n_clusters)
plt.text(-4, 13, text, size=15)
plt.show()
```





6 Clusters

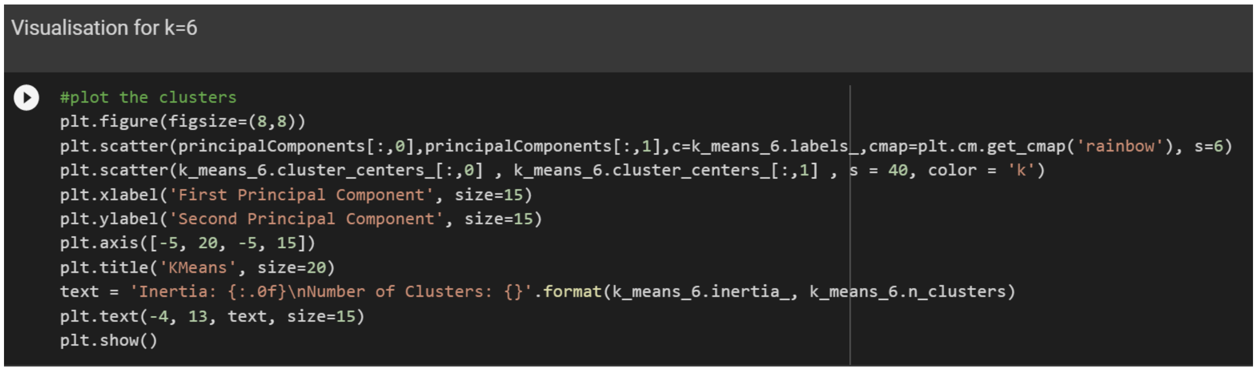
Fit a kmeans model with 6 clusters

```
[63] # k means using 6 clusters and k-means++ initialization
     n_clusters = 6
     k_means_6 = KMeans( n_clusters = n_clusters, init='k-means++')
     k_means_6.fit(principalComponents)
     pred = k_means_6.predict(principalComponents)
```

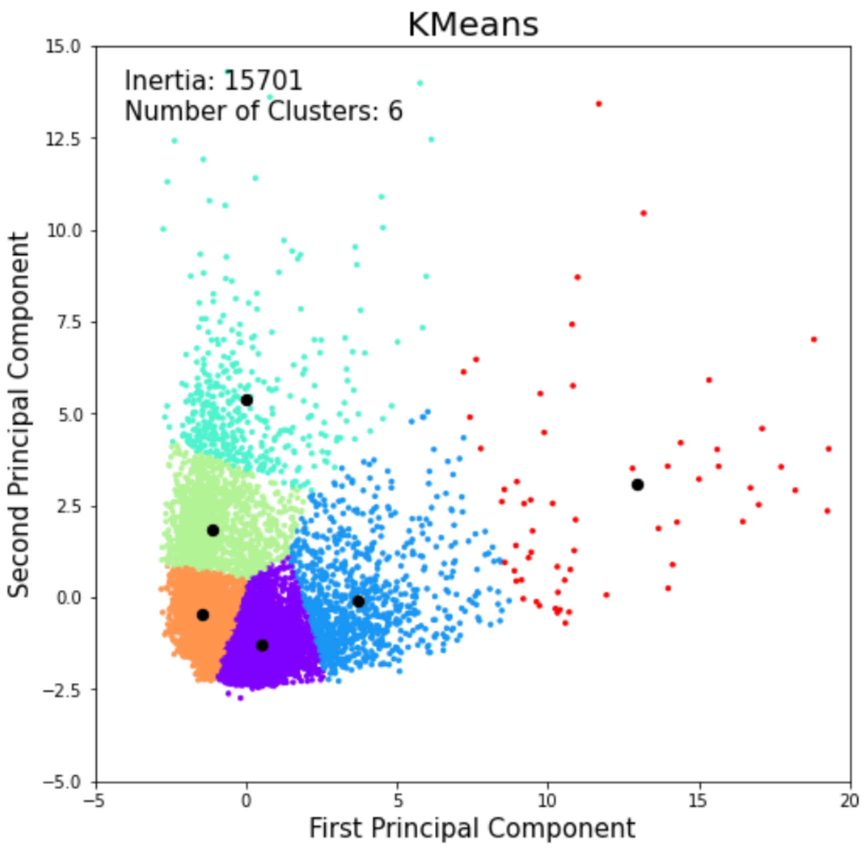
Get value counts for 6 clusters

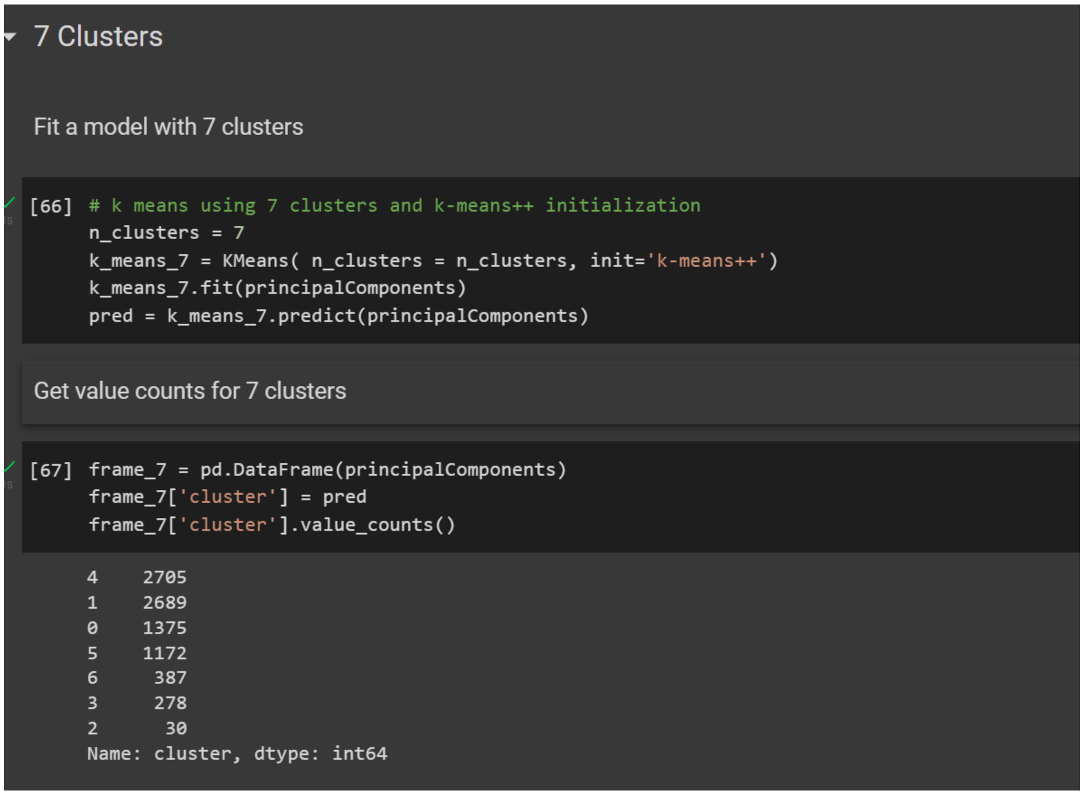
```
[64] frame_6 = pd.DataFrame(principalComponents)
     frame_6['cluster'] = pred
     frame_6['cluster'].value_counts()
```

```
0    2903
4    2875
3    1492
1     906
2     395
5      65
Name: cluster, dtype: int64
```


Visualisation for k=6

```
#plot the clusters
plt.figure(figsize=(8,8))
plt.scatter(principalComponents[:,0],principalComponents[:,1],c=k_means_6.labels_,cmap=plt.cm.get_cmap('rainbow'), s=6)
plt.scatter(k_means_6.cluster_centers_[:,0] , k_means_6.cluster_centers_[:,1] , s = 40, color = 'k')
plt.xlabel('First Principal Component', size=15)
plt.ylabel('Second Principal Component', size=15)
plt.axis([-5, 20, -5, 15])
plt.title('KMeans', size=20)
text = 'Inertia: {:.0f}\nNumber of Clusters: {}'.format(k_means_6.inertia_, k_means_6.n_clusters)
plt.text(-4, 13, text, size=15)
plt.show()
```





7 Clusters

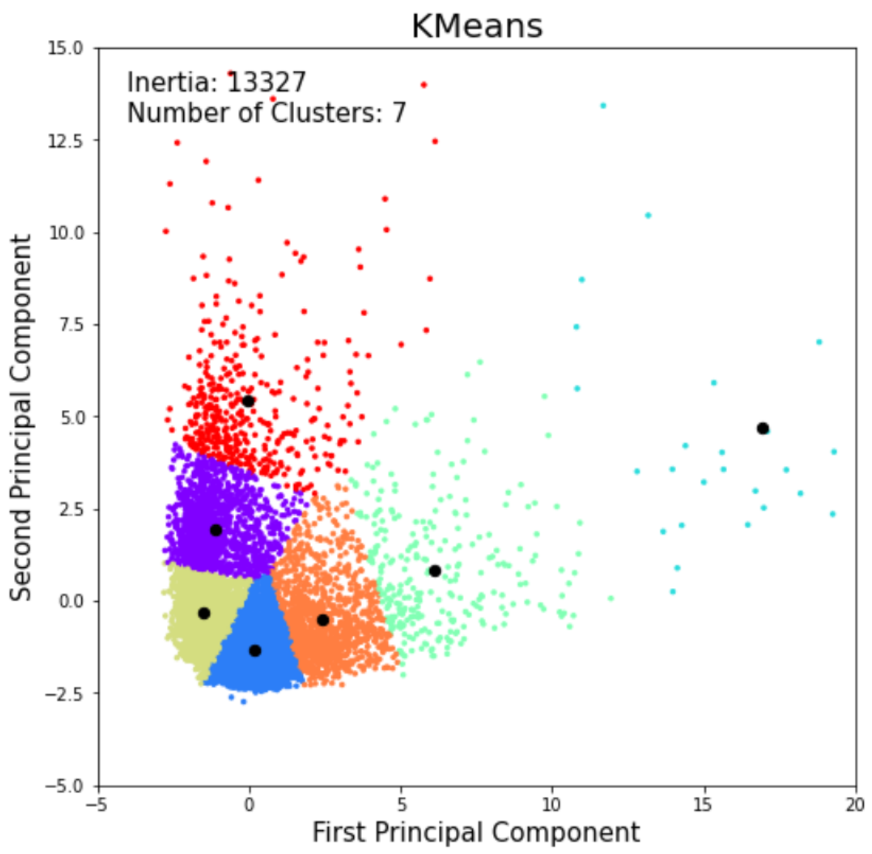
Fit a model with 7 clusters

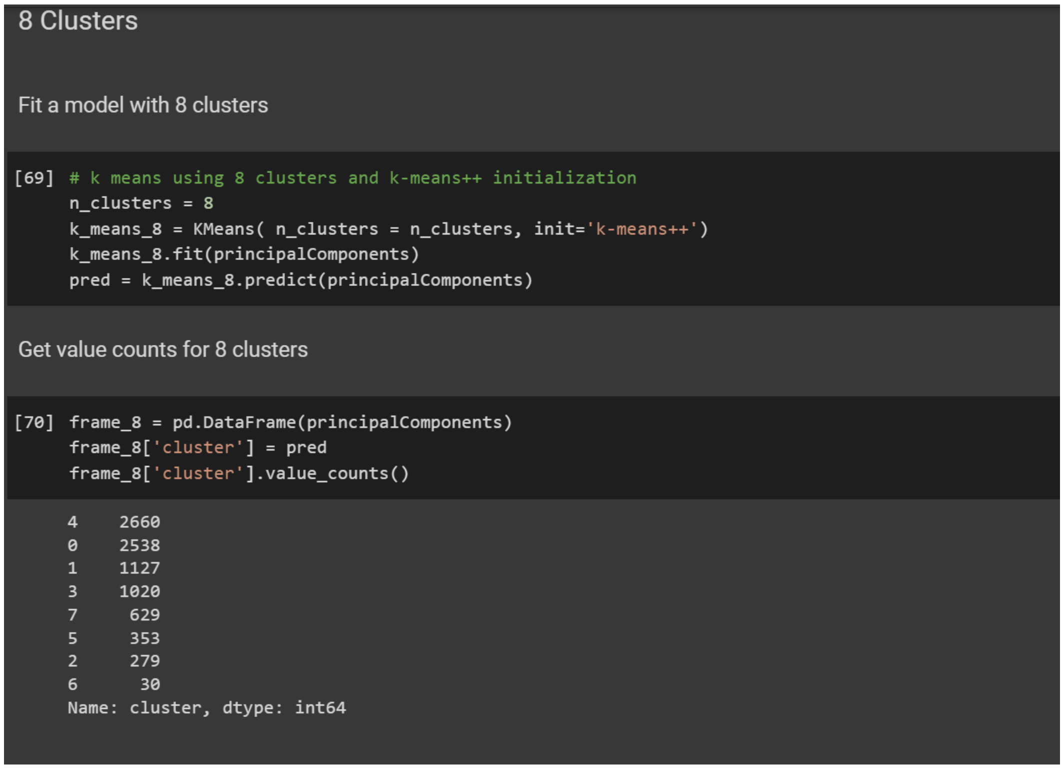
```
[66] # k means using 7 clusters and k-means++ initialization
     n_clusters = 7
     k_means_7 = KMeans( n_clusters = n_clusters, init='k-means++')
     k_means_7.fit(principalComponents)
     pred = k_means_7.predict(principalComponents)
```

Get value counts for 7 clusters

```
[67] frame_7 = pd.DataFrame(principalComponents)
     frame_7['cluster'] = pred
     frame_7['cluster'].value_counts()
```

```
4    2705
1    2689
0    1375
5    1172
6     387
3     278
2      30
Name: cluster, dtype: int64
```




8 Clusters

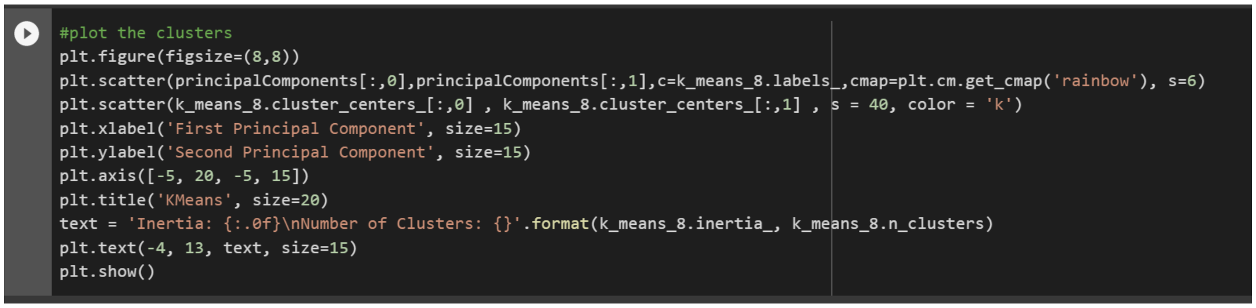
Fit a model with 8 clusters

```
[69] # k means using 8 clusters and k-means++ initialization
     n_clusters = 8
     k_means_8 = KMeans( n_clusters = n_clusters, init='k-means++')
     k_means_8.fit(principalComponents)
     pred = k_means_8.predict(principalComponents)
```

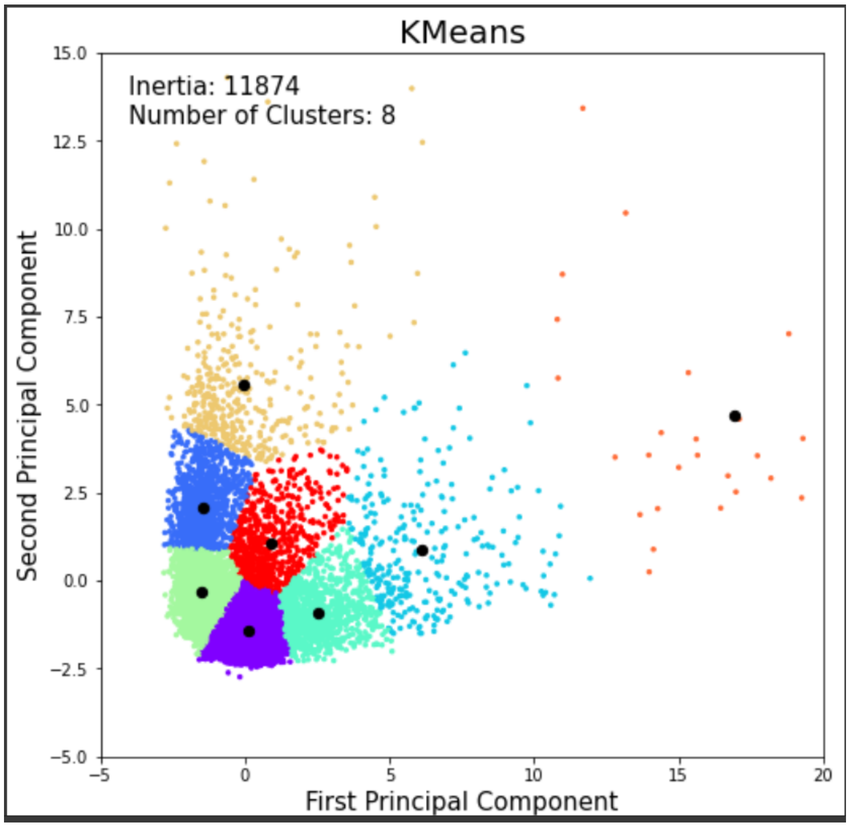
Get value counts for 8 clusters

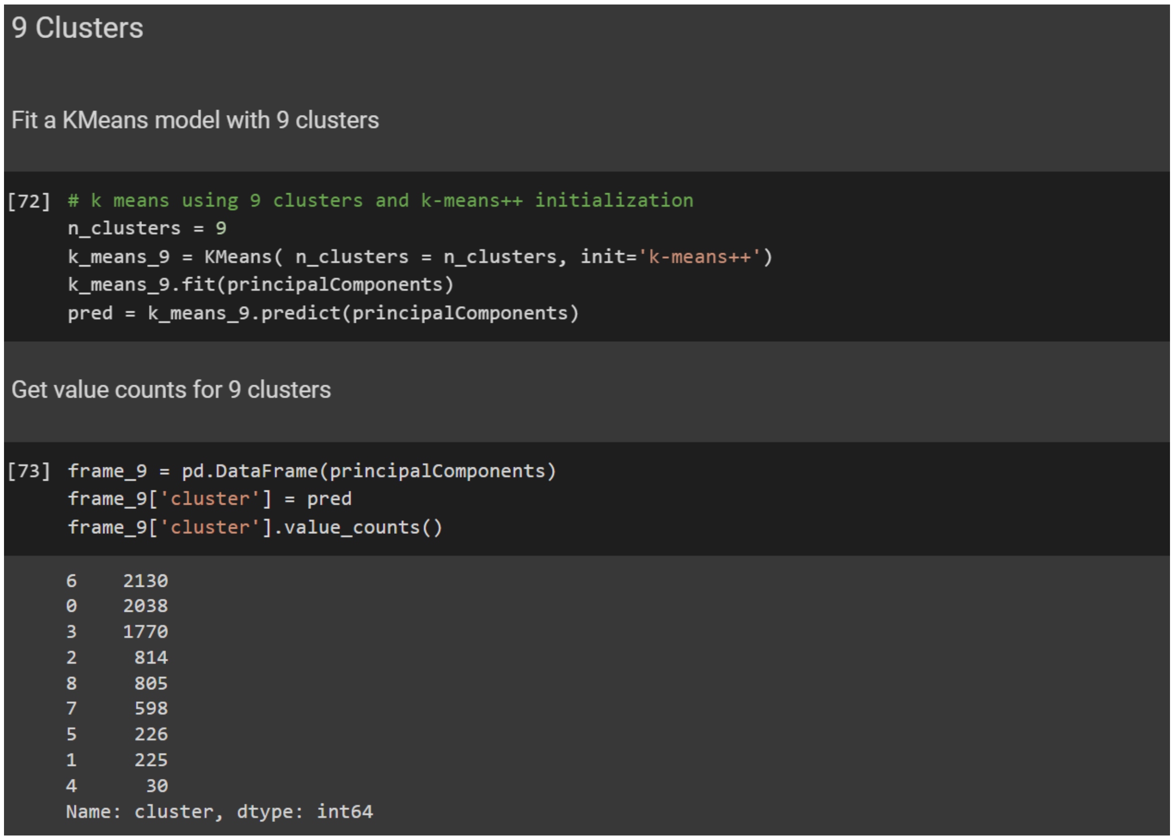
```
[70] frame_8 = pd.DataFrame(principalComponents)
     frame_8['cluster'] = pred
     frame_8['cluster'].value_counts()
```

```
4    2660
0    2538
1    1127
3    1020
7     629
5     353
2     279
6      30
Name: cluster, dtype: int64
```



```
#plot the clusters
plt.figure(figsize=(8,8))
plt.scatter(principalComponents[:,0],principalComponents[:,1],c=k_means_8.labels_,cmap=plt.cm.get_cmap('rainbow'), s=6)
plt.scatter(k_means_8.cluster_centers_[:,0] , k_means_8.cluster_centers_[:,1] , s = 40, color = 'k')
plt.xlabel('First Principal Component', size=15)
plt.ylabel('Second Principal Component', size=15)
plt.axis([-5, 20, -5, 15])
plt.title('KMeans', size=20)
text = 'Inertia: {:.0f}\nNumber of Clusters: {}'.format(k_means_8.inertia_, k_means_8.n_clusters)
plt.text(-4, 13, text, size=15)
plt.show()
```





9 Clusters

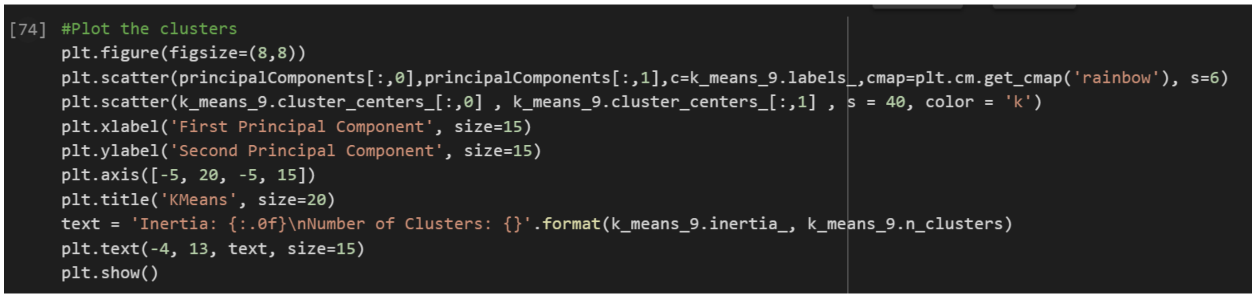
Fit a KMeans model with 9 clusters

```
[72] # k means using 9 clusters and k-means++ initialization
     n_clusters = 9
     k_means_9 = KMeans( n_clusters = n_clusters, init='k-means++')
     k_means_9.fit(principalComponents)
     pred = k_means_9.predict(principalComponents)
```

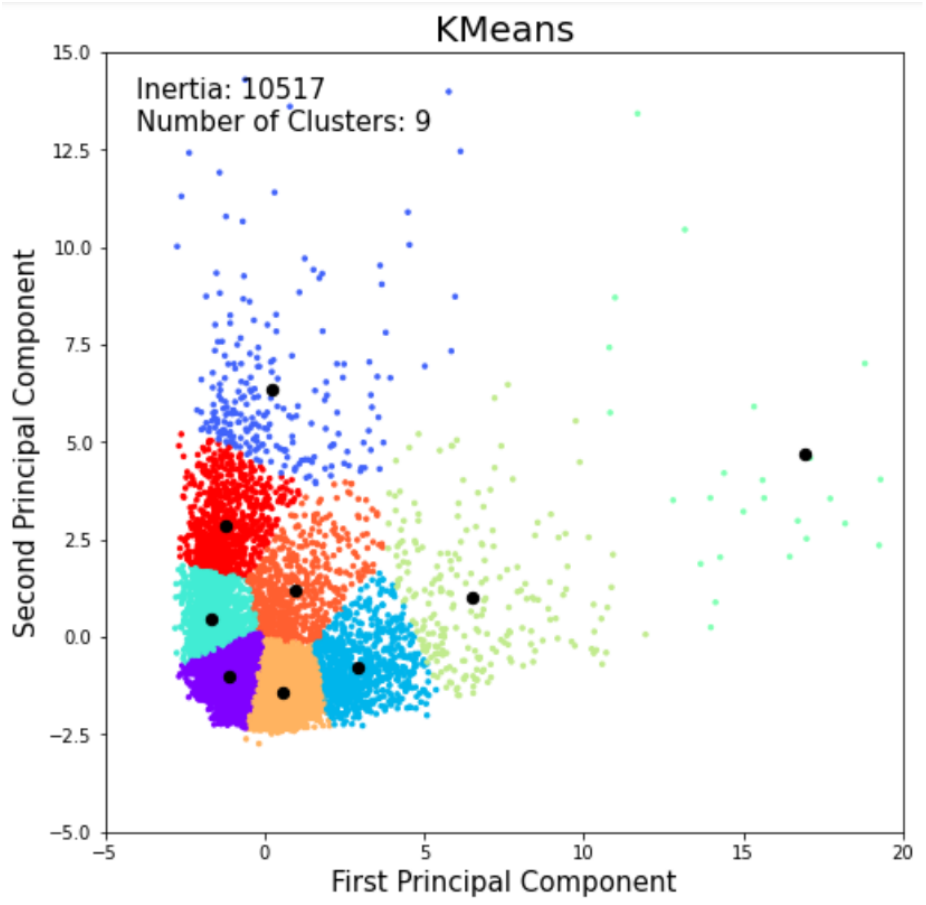
Get value counts for 9 clusters

```
[73] frame_9 = pd.DataFrame(principalComponents)
     frame_9['cluster'] = pred
     frame_9['cluster'].value_counts()
```

```
     6    2130
     0    2038
     3    1770
     2     814
     8     805
     7     598
     5     226
     1     225
     4      30
     Name: cluster, dtype: int64
```



```
[74] #Plot the clusters
     plt.figure(figsize=(8,8))
     plt.scatter(principalComponents[:,0],principalComponents[:,1],c=k_means_9.labels_,cmap=plt.cm.get_cmap('rainbow'), s=6)
     plt.scatter(k_means_9.cluster_centers_[:,0] , k_means_9.cluster_centers_[:,1] , s = 40, color = 'k')
     plt.xlabel('First Principal Component', size=15)
     plt.ylabel('Second Principal Component', size=15)
     plt.axis([-5, 20, -5, 15])
     plt.title('KMeans', size=20)
     text = 'Inertia: {:.0f}\nNumber of Clusters: {}'.format(k_means_9.inertia_, k_means_9.n_clusters)
     plt.text(-4, 13, text, size=15)
     plt.show()
```


KMeans
Inertia: 10517
Number of Clusters: 9
Second Principal Component
First Principal Component