K-Means Clustering: A Tutorial in R and Python

Today we are taking a look at the K-means clustering algorithm in both the R and Python languages. We'll to break down the algorithm so that the everyday business professional can understand and use this algorithm to group together similar customers, patients, texts, images, or anything you want to try to break into similar groups.

What exactly is K-means?

K-means is an unsupervised clustering algorithm. Unsupervised means that there is no response variable, outcome, Y variable, or label (all of these words mean the same thing). Clustering means that our goal for the algorithm is to group our observations. What does the K mean (pun intended)? K is the number of clusters that we are going to group our data into. Without the need for a response we can classify almost any data set. But here's the big catch with unsupervised learning: without having a response or a way to label the data there is no way to check the accuracy. Instead we rely on metrics for how similar observations are within each cluster, and how different they are across the clusters.

A Simple Business Example

Here's a business example to provide better clarity. Suppose we had a bunch of data about customers from a video streaming website. If we wanted to classify the customers into groups based on if they would stream the horror genre of movies to target those customers for some upcoming halloween promotional deal, then we would use a classification method. However, in this case we have a response: Whether the customer watched horror movies or not. So we would use a supervised classification method such as logistic regression. We would run our algorithm and check if yes, our algorithm classified the customer correctly based on whether or not the customer actually did watch horror movies.

But this is **NOT** what we do for K-means. If we wanted to use K-means or another unsupervised classification algorithm we would not worry about whether the customer had watched horror movies or not (the response). We would use the algorithm on all of the data (including the horror movies or not variable) and see which observations would be grouped together by the algorithm.

The two big questions you probably have our how and why we would do this. The how we will go into in much greater detail in the algorithm section but on a very high level K-means groups points that are close in distance. The why is that k-means could cluster our data in ways that we might not have considered. In our theoretical example, perhaps one of the k-means clusters is a group of customers that happens to spend money on promotional deals. So by sending those customers our halloween promotional deal we could make the company a lot of money. Or it provides us with another customer target group to send our promotion to in addition to the results from our logistic regression. Conversely, sending our promotion to those customers might not net the company any profit at all.

As you can see from our example, there's no way to determine if the algorithm clusters accurately. The key to unsupervised learning in general is domain knowledge - after we have

clustered our customers we may be able to look closer at each group to gain some insight into difference across clusters. K-means also really shines with data sets where there is not a choice of unsupervised vs. supervised learning as you will see in our example below where there is no clear choice for a response.

Data:

The data set used for this tutorial is from Kaggle and can be found here: https://www.kaggle.com/arjunbhasin2013/ccdata**.** This data set contains information for nearly 9000 credit card customers with information such as tenure, credit limit, purchases, etc. Since there is not an outcome variable we are trying to predict with this dataset, it makes an excellent candidate for unsupervised learning methods such as K-means clustering. For example, the credit card company may want to cluster their customer database to target new products based on their current and past information.

As a first step of cleaning the data, we noticed that there was one NA in the credit limit field and 313 NAs in the Minimum_Payments field. Two common ways to handle such instances in data sets are to either replace the NA with the mean of the particular field, or just to drop the observations altogether. Since there is no right or wrong answer, we explore both options by dropping the NA from our R code and imputing the missing values in our Python code. How to handle missing values in your own data set will be a decision you'll have to make depending on your particular situation.

The final step for cleaning out data is to scale each field using the scale() function so that they have a mean of 0 and variance of 1. This step is not required in order to run K-means clustering but the following questions should be considered when deciding whether or not to scale:

- Do I want all of my variables to have equal weight regardless of variance?
- Do I want the variables with the largest variance to drive clusters?
- Are my variables on the same scale? Do I want equal importance regardless of scale?

There is no right or wrong answer to any of these questions, but they should be considered when deciding whether or not to scale variables. In our dataset, we decided to scale based on two factors:

1. Each of our variables had very different scales and units
2. Given a lack of specific domain knowledge to indicate otherwise, we wanted all variables to have equal weight in our clusters.

The K-Means Algorithm

The Goal for K-Means Algorithm: $$minimize\ C_1, ...C_K \left\{ \sum_{k=1}^{K} \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2 \right\}$$

What this says in words is that we want to minimize the within cluster variation. The within cluster variation for the (k^{th}) cluster is defined (most commonly and specifically in our

example) as the sum of all pairwise distances between observations in the cluster divided by total number of observations in the cluster. Other metrics besides distance are possible to use but that is the most common and the one we will use in this example. While that still sounds highly technical, walking through the steps should shed some more light on the algorithm

Step 1:

Randomly assign a number, $1, ..., K$ to each observation (input variable or x_i).

Step 2:

For each $1, ..., K$ clusters compute the center of the cluster, commonly called the centroid. This is calculated as the mean (average) of the observations in the k^{th} cluster.

Step 3:

Reassign each observation to the closest centroid. Where closest is determined by squared euclidean distance (a.k.a. The distance formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$)

Step 4:

Repeat steps 2 and 3 until the cluster centers stop changing.

Graphical Illustration of K-Means Algorithm

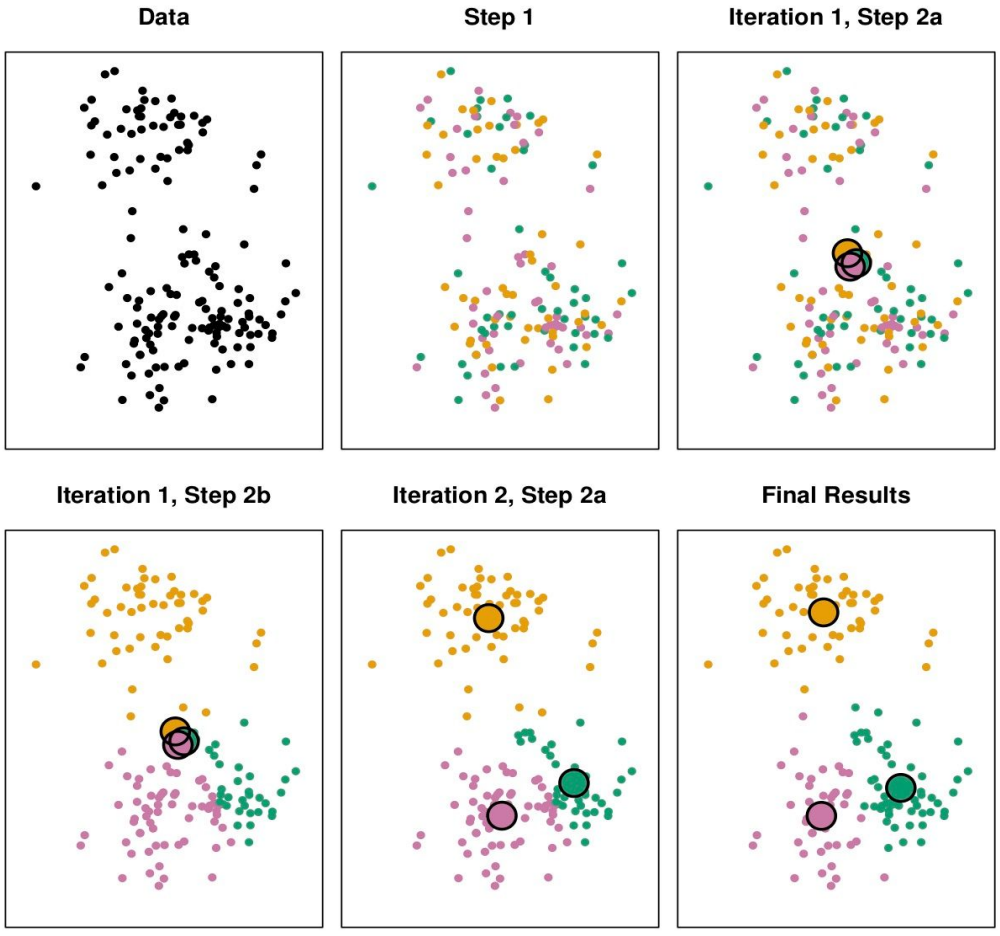


Figure 1. K-Means Plot (from James, 2013, p. 390)

Figure 1 shows a graphical representation of the algorithm. Step 1 in the top center panel corresponds with Step 1 above where we randomly assign each observation to a cluster denoted by color. Iteration 1, Step 2a in the top right corresponds with Step 2 above. Here the centroids are computed as the average of the observation in each cluster. Another way to put it would be the average of all the yellow observations is the big yellow circle, and so on for the other colors. Iteration 1, Step 2b in the bottom left corner corresponds to Step 3 above, where observations are reassigned to a new cluster based on how close they are to the new centroids. Graphically, this means the colors of observations are reassigned to the color of the nearest big circle. The bottom center and right graphs represent Step 4, where steps 2 and 3 are repeated until the centroids no longer change.

Implementing K-means: Below you can find links for how to implement the method described above in both R and Python

Click here for R Code:
Click here for Python:
(These would be hyperlinks on a webpage but for the project see below)

What Next?

Now that you have you have your clusters, the next steps are really up to your business. After diving deeper to understanding the differences across clusters you will be able to target ads, promotions, or particular products with the hopes of saving costs and increasing revenue in the process.

References

Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. (2013). An introduction to statistical learning : with applications in R. New York :Springer,

How to Implement K-Means Clustering in R

Geting Started

First we need to load the necessary package to conduct the analysis.

```
library(factoextra)
```

Cleaning Data

As noted earlier, all of the rows with "NA" for any factor were removed. Additionally, we removed the first field which just contains customer ID numbers. You can do so using the following code, using summary() to confirm these have been removed:

```
ccDat<-read.csv("CC GENERAL.csv")  ## Read in the Data
which(is.na(ccDat$MINIMUM_PAYMENTS)) ## Looking for NA's
which(is.na(ccDat$CREDIT_LIMIT))
## Now Remove NA's and Customer ID  column
ccDat<-ccDat[-5204,]
ccDat<-ccDat[-c(which(is.na(ccDat$MINIMUM_PAYMENTS))),]
ccDat<-ccDat[,-1]
summary(ccDat)
```

To Scale or Not to Scale?

As discussed previously, this is always a challenging question to answer and there is definitely not a right or wrong answer. However, for reasones mentioned in the intro we move forward with scaling.

```
ccDat_Std <- data.frame(scale(ccDat))
```

Clustering the Data

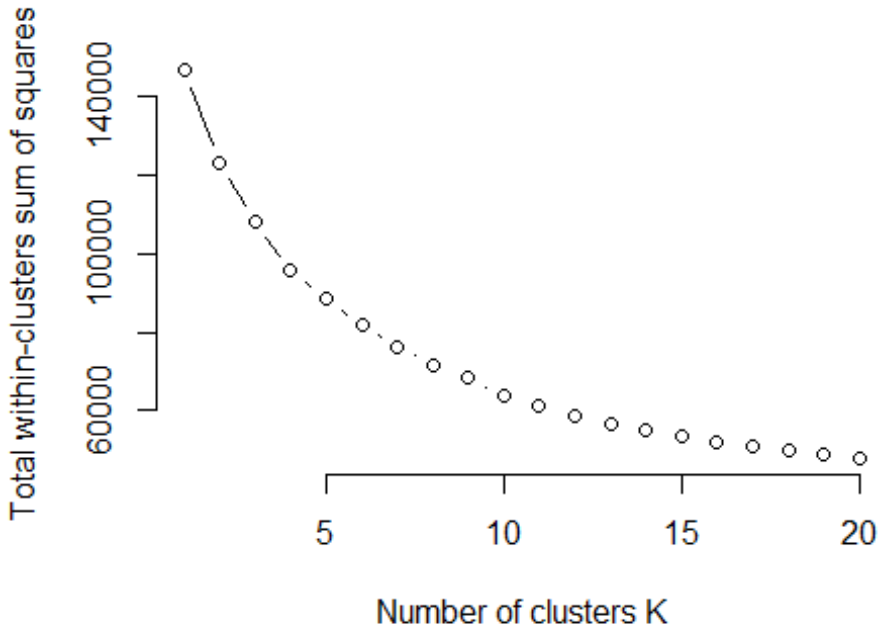
We are now ready to actually implement the clustering algorithim. To do so, we use the Kmeans() function. For this example, there are three parameters for the function. The first is just our dataset, 'ccDat_STD.' Next, we need to input the number of clusters we want to use. At this point in the analysis, we are not sure what that number should be so we will loop through K = 1 to K = 20 using the sapply() function. The final parameter for Kmeans() is 'nstart.' This value is used in the first step of the k-means algorithim when we are assigning each observation to a random cluster. In order to ensure we are getting the best clusters, we often will want to run the algorithim with many random starts and choose the best as our final cluster assignments. The 'nstart' parameter does just this, with the funciton returning the best cluster out of those attempts. There is no perfect number of starts to use to balance run time and strength of final clusters, but 20 should be sufficient in most cases.

```
set.seed(123)
wss <- sapply(1:20, function(k){kmeans(ccDat_Std, k, nstart=20)$tot.withinss}
)
```

Selecting K

We now plot the total within cluster sum of squares to help us determine the value for "K" we should select. To do this we follow the "elbow method," which means we are looking for a sharp change in the amount that the total within cluster sum of squares decreases with the inclusion of an additional cluster. We can see from the graph below that this happens after K=4 (the decrease from K = 3 to K = 4 is steeper than the decrease from K = 4 to K= 5). The elbow method can be helpful for determining the number of clusters when you don't have specific insights from your business. However, you may have a specific number of clusters in mind before this analysis, in which case this step would be unnecessary.

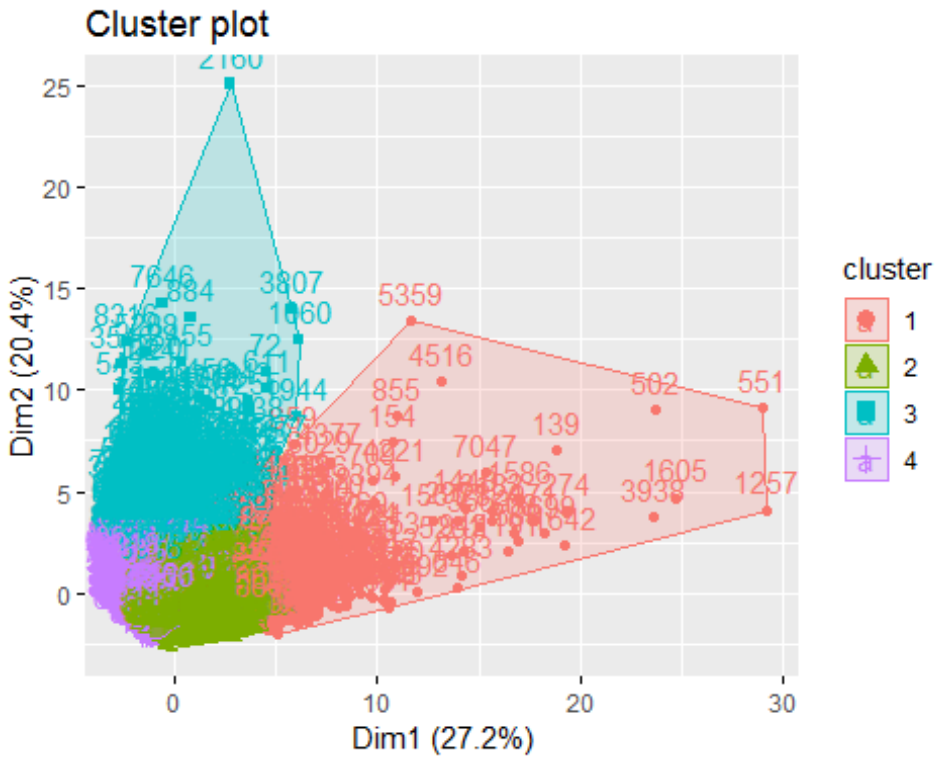
```
plot(1:20, wss[1:20],
     type="b", pch = 1, frame = FALSE,
     xlab="Number of clusters K",
     ylab="Total within-clusters sum of squares")
```



Visualizing Final Model

Now that we have determined the number of clusters for our final model, we may want to visualize how the clustering is working. However, given the dimensionality of our space we only plot the first two prinicpal components or else we would need a plot for each pair of predictors.

```
finalkmeans <- kmeans(ccDat_Std, 4, nstart=20)
fviz_cluster(finalkmeans, data = ccDat_Std)
```



Analyzing the final model

By simply typing the name of the model, 'finalkmeans,' we are able to see the mean of each cluster for all 17 variables, as well as determine which cluster each individual observation is in. For example, we see that the mean balance is highest in our third cluster, while the mean tenure is highest in our first cluster. Diving deeper into these sorts of descriptive metrics of the cluster is likely where your business will find the most value from this unsupervised method, as products or advertising can be differentiated across the clusters.

```
finalkmeans

## K-means clustering with 4 clusters of sizes 391, 3276, 1155, 3814
##
## Cluster means:
##      BALANCE BALANCE_FREQUENCY  PURCHASES ONEOFF_PURCHASES
## 1  0.9486524         0.4460078  3.1460718      2.744816577
## 2 -0.3262022         0.2468721  0.1126475      0.004264522
## 3  1.4570937         0.3570119 -0.2403425     -0.169600023
## 4 -0.2583188        -0.3658865 -0.3465002     -0.233693191
##   INSTALLMENTS_PURCHASES CASH_ADVANCE PURCHASES_FREQUENCY
## 1              2.3941088   -0.1559753           1.1200438
## 2              0.2582812   -0.3653530           0.9771706
## 3             -0.2566518    1.6943725          -0.5129989
```

```
## 4             -0.3895629   -0.1833030          -0.7988029
##   ONEOFF_PURCHASES_FREQUENCY PURCHASES_INSTALLMENTS_FREQUENCY
## 1                  1.7843074                        1.0494916
## 2                  0.3222531                        0.8691288
## 3                 -0.2206328                       -0.4552502
## 4                 -0.3929036                       -0.7162568
##   CASH_ADVANCE_FREQUENCY CASH_ADVANCE_TRX PURCHASES_TRX CREDIT_LIMIT
## 1            -0.32820699       -0.1734290     3.0104937   1.43095846
## 2            -0.46456538       -0.3613717     0.2977302  -0.06928368
## 3             1.73410714        1.6120383    -0.2891468   0.85427789
## 4            -0.09246162       -0.1599999    -0.4767967  -0.34588946
##     PAYMENTS MINIMUM_PAYMENTS PRC_FULL_PAYMENT      TENURE
## 1  1.9518869       0.48076171        0.4462792  0.31614291
## 2 -0.1386313      -0.08902596        0.3881404  0.05316575
## 3  0.6090544       0.49496706       -0.4183565 -0.11373867
## 4 -0.2654666      -0.12270995       -0.2524492 -0.04363260
##
## Clustering vector:
##    1    2    3    5    6    7    8    9   10   11   12   13   14   15   16
##    4    3    2    4    2    1    2    4    4    2    4    2    2    4    3


...


## 8938 8939 8940 8941 8942 8943 8944 8946 8948 8949 8950
##    4    4    4    2    3    2    4    2    2    4    4
##
## Within cluster sum of squares by cluster:
## [1] 20727.05 28462.33 22618.44 23827.45
##  (between_SS / total_SS =  34.9 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

K-Means-Edited

November 15, 2018

1 Implement K-Means Clustering Algorithm in Python Based on Credit Card Dataset

In this case, we try to use the K-means method to build a credit card user segmentation. ## Getting Started First, we need to load the necessary libraries. The first section contains the basic libraries. `Pandas` provides us the tool to manipulate with the data. `Numpy` gives us some basic statistical tools. The rest are the visualization tools.

```
In [1]: import pandas as pd
        import matplotlib.pyplot as plt
        import numpy as np
        import matplotlib as mpl
        import seaborn as sns

        %matplotlib inline
        plt.style.use('seaborn-white')
```

The second section is calling the basic libraries to tango with our machine learning algorithm. `KMeans` is our key method so we label it as a shortcut.

```
In [2]: from sklearn.preprocessing import scale
        from sklearn.cluster import KMeans
        from scipy.cluster import hierarchy
```

1.1 Load the dataset

The dataset contains the usage behavior of about 9000 active credit card holders during the last 6 months. Each observed holder has 18 features. ### Getting the basic description of the data

```
In [3]: data=pd.read_csv('./CC GENERAL.csv')

In [4]: data.describe()

Out[4]:              BALANCE  BALANCE_FREQUENCY     PURCHASES  ONEOFF_PURCHASES  \
        count    8950.000000        8950.000000   8950.000000       8950.000000
        mean     1564.474828           0.877271   1003.204834        592.437371
        std      2081.531879           0.236904   2136.634782       1659.887917
        min         0.000000           0.000000      0.000000          0.000000
```

25%	128.281915	0.888889	39.635000	0.000000
50%	873.385231	1.000000	361.280000	38.000000
75%	2054.140036	1.000000	1110.130000	577.405000
max	19043.138560	1.000000	49039.570000	40761.250000

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	\
count	8950.000000	8950.000000	8950.000000	
mean	411.067645	978.871112	0.490351	
std	904.338115	2097.163877	0.401371	
min	0.000000	0.000000	0.000000	
25%	0.000000	0.000000	0.083333	
50%	89.000000	0.000000	0.500000	
75%	468.637500	1113.821139	0.916667	
max	22500.000000	47137.211760	1.000000	

	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY	\
count	8950.000000	8950.000000	
mean	0.202458	0.364437	
std	0.298336	0.397448	
min	0.000000	0.000000	
25%	0.000000	0.000000	
50%	0.083333	0.166667	
75%	0.300000	0.750000	
max	1.000000	1.000000	

	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX	PURCHASES_TRX	CREDIT_LIMIT	\
count	8950.000000	8950.000000	8950.000000	8949.000000	
mean	0.135144	3.248827	14.709832	4494.449450	
std	0.200121	6.824647	24.857649	3638.815725	
min	0.000000	0.000000	0.000000	50.000000	
25%	0.000000	0.000000	1.000000	1600.000000	
50%	0.000000	0.000000	7.000000	3000.000000	
75%	0.222222	4.000000	17.000000	6500.000000	
max	1.500000	123.000000	358.000000	30000.000000	

	PAYMENTS	MINIMUM_PAYMENTS	PRC_FULL_PAYMENT	TENURE
count	8950.000000	8637.000000	8950.000000	8950.000000
mean	1733.143852	864.206542	0.153715	11.517318
std	2895.063757	2372.446607	0.292499	1.338331
min	0.000000	0.019163	0.000000	6.000000
25%	383.276166	169.123707	0.000000	12.000000
50%	856.901546	312.343947	0.000000	12.000000
75%	1901.134317	825.485459	0.142857	12.000000
max	50721.483360	76406.207520	1.000000	12.000000

1.1.1 Handling with the missing values

First, we need to see whether the data is cleaned or not. Only the `MINiMUM_PAYMENTS` column contains 313 NA.

```
In [5]: missing=data.isna().sum()
```

```
In [6]: print(missing)
```

```
CUST_ID                             0
BALANCE                             0
BALANCE_FREQUENCY                   0
PURCHASES                           0
ONEOFF_PURCHASES                    0
INSTALLMENTS_PURCHASES              0
CASH_ADVANCE                        0
PURCHASES_FREQUENCY                 0
ONEOFF_PURCHASES_FREQUENCY          0
PURCHASES_INSTALLMENTS_FREQUENCY    0
CASH_ADVANCE_FREQUENCY              0
CASH_ADVANCE_TRX                    0
PURCHASES_TRX                       0
CREDIT_LIMIT                        1
PAYMENTS                            0
MINIMUM_PAYMENTS                  313
PRC_FULL_PAYMENT                    0
TENURE                              0
dtype: int64
```

My fellow cohorts have already used some methods to drop the NAs. Instead of removing them, I try to refill them with column means.

```
In [7]: data=data.fillna(data.mean())
```

1.1.2 Scale the data

As discussed previously, this is always a challenging question to answer and there is definitely not a right or wrong answer. However, given the particular example we are studying we move forward with scaling!

```
In [8]: #Removing the ID column
        data_std=data.iloc[:,1:]
```

Notice in the describe section that the mean is now zero and standard deviation is now one.

```
In [9]: data_std=pd.DataFrame(scale(data_std), columns=data_std.columns)
```

```
/Users/dingxuanzhang/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:1: DataConversi
  """Entry point for launching an IPython kernel.
```

In [10]:
```
data_std.describe()
```

Out[10]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES
count	8.950000e+03	8.950000e+03	8.950000e+03	8.950000e+03
mean	4.195651e-16	1.209548e-14	-9.278518e-16	3.916048e-15
std	1.000056e+00	1.000056e+00	1.000056e+00	1.000056e+00
min	-7.516398e-01	-3.703271e+00	-4.695519e-01	-3.569340e-01
25%	-6.900078e-01	4.904486e-02	-4.510006e-01	-3.569340e-01
50%	-3.320286e-01	5.180838e-01	-3.004541e-01	-3.340396e-01
75%	2.352559e-01	5.180838e-01	5.004652e-02	-9.056763e-03
max	8.397489e+00	5.180838e-01	2.248351e+01	2.420107e+01

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY
count	8.950000e+03	8.950000e+03	8.950000e+03
mean	2.275424e-15	5.693056e-15	-3.716084e-16
std	1.000056e+00	1.000056e+00	1.000056e+00
min	-4.545762e-01	-4.667856e-01	-1.221758e+00
25%	-4.545762e-01	-4.667856e-01	-1.014125e+00
50%	-3.561562e-01	-4.667856e-01	2.404259e-02
75%	6.366321e-02	6.435242e-02	1.062211e+00
max	2.442689e+01	2.201112e+01	1.269843e+00

	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY
count	8.950000e+03	8.950000e+03
mean	2.021052e-15	2.353921e-16
std	1.000056e+00	1.000056e+00
min	-6.786608e-01	-9.169952e-01
25%	-6.786608e-01	-9.169952e-01
50%	-3.993193e-01	-4.976286e-01
75%	3.269728e-01	9.701506e-01
max	2.673451e+00	1.599199e+00

	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX	PURCHASES_TRX	CREDIT_LIMIT
count	8.950000e+03	8.950000e+03	8.950000e+03	8.950000e+03
mean	5.729495e-16	-7.151821e-16	-3.570713e-15	-5.142466e-15
std	1.000056e+00	1.000056e+00	1.000056e+00	1.000056e+00
min	-6.753489e-01	-4.760698e-01	-5.917959e-01	-1.221536e+00
25%	-6.753489e-01	-4.760698e-01	-5.515646e-01	-7.955261e-01
50%	-6.753489e-01	-4.760698e-01	-3.101767e-01	-4.107426e-01
75%	4.351492e-01	1.100739e-01	9.213645e-02	5.512163e-01
max	6.820521e+00	1.754785e+01	1.381101e+01	7.010083e+00

	PAYMENTS	MINIMUM_PAYMENTS	PRC_FULL_PAYMENT	TENURE
count	8.950000e+03	8.950000e+03	8.950000e+03	8.950000e+03
mean	-2.421651e-16	1.863190e-17	-2.201566e-15	1.556446e-14
std	1.000056e+00	1.000056e+00	1.000056e+00	1.000056e+00
min	-5.986883e-01	-3.708230e-01	-5.255510e-01	-4.122768e+00
25%	-4.662913e-01	-2.975162e-01	-5.255510e-01	3.606795e-01

```
50%   -3.026846e-01     -2.268130e-01     -5.255510e-01  3.606795e-01
75%    5.802976e-02     -5.366135e-16     -3.712234e-02  3.606795e-01
max    1.692228e+01      3.241509e+01      2.893453e+00  3.606795e-01
```

1.2 K-Means Implementation

We'll use the elbow method to find a good number of clusters with the KMeans++ algorithm.

```
In [11]: value=data_std.values
```

```
In [12]: #Use the Elbow method to find a good number of clusters
         elbow=[]
         for ii in range(1, 30):
             kmeans=KMeans(n_clusters=ii, init='k-means++', n_init=10, max_iter=300)
             kmeans.fit_predict(value)
             elbow.append(kmeans.inertia_)
```

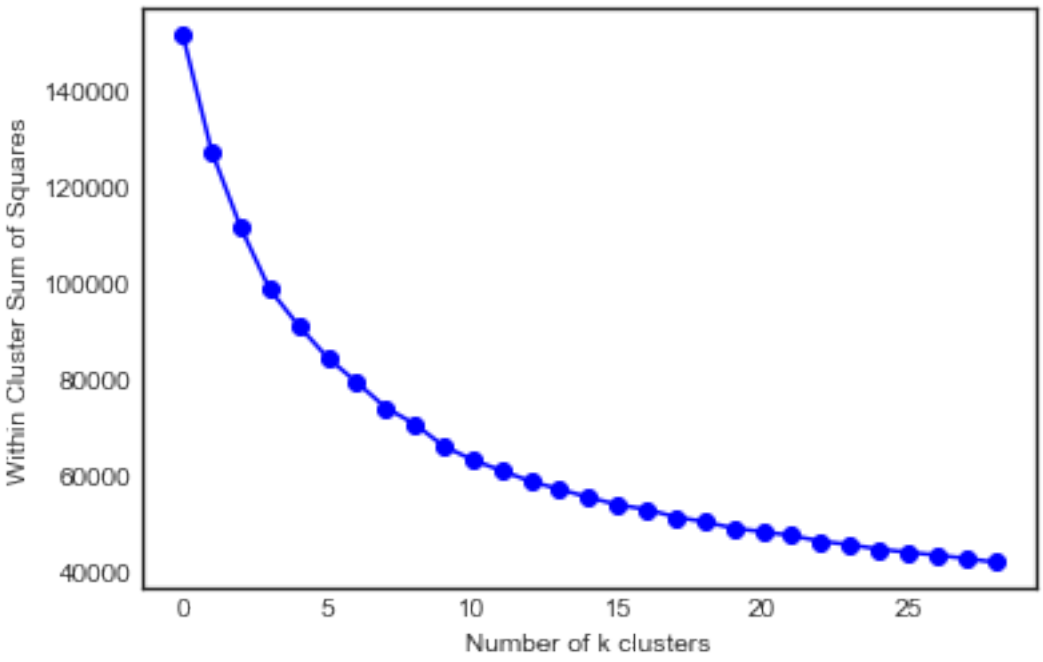
The method we used above is to minimize the within-cluster sum of squares. We try to loop through 1 to 29, hoping to find the optimal k. The default parameter `n_init` is 10, which means 'Number of time the k-means algorithm will be run with different centroid seeds'. Maximum iteration is the default number 300.

1.2.1 Choosing k using Elbow method

We now plot the total within cluster sum of squares to help us determine the value for "K" we should select. To do this we follow the "elbow method," which means we are looking for a sharp change in the amount that the total within cluster sum of squares decreases with the addition of an additional cluster.

```
In [13]: #Setting the elbow plot
         plt.plot(elbow, 'bo-', label="ELBOW")
         plt.xlabel("Number of k clusters")
         plt.ylabel("Within Cluster Sum of Squares")
```

```
Out[13]: Text(0, 0.5, 'Within Cluster Sum of Squares')
```



The problem with our K Means Elbow Plot is that the elbow is very ambiguous. While in R we tried K=4, here we will try K=9.

```
In [14]: # try k=9
         kmeans = KMeans(n_clusters=9, init="k-means++", n_init=10, max_iter=300)

In [15]: kmeans.fit(value)

Out[15]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
             n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
             random_state=None, tol=0.0001, verbose=0)

In [16]: kmeans.inertia_

Out[16]: 69979.11172076217
```

As it's difficult to visualise clusters when the data is high-dimensional, we will call the `pairplot` in the seaborn library.

```
In [17]: Pred=kmeans.fit_predict(value)

In [18]: Pred=pd.Series(Pred)

In [19]: #Add cluster label to our original scaled dataset
         data_std['cluster']=Pred

In [20]: data_std.head()
```

Out[20]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES
0	-0.731989	-0.249434	-0.424900	-0.356934
1	0.786961	0.134325	-0.469552	-0.356934
2	0.447135	0.518084	-0.107668	0.108889
3	0.049099	-1.016953	0.232058	0.546189
4	-0.358775	0.518084	-0.462063	-0.347294

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY
0	-0.349079	-0.466786	-0.806490
1	-0.454576	2.605605	-1.221758
2	-0.454576	-0.466786	1.269843
3	-0.454576	-0.368653	-1.014125
4	-0.454576	-0.466786	-1.014125

	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY
0	-0.678661	-0.707313
1	-0.678661	-0.916995
2	2.673451	-0.916995
3	-0.399319	-0.916995
4	-0.399319	-0.916995

	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX	PURCHASES_TRX	CREDIT_LIMIT
0	-0.675349	-0.476070	-0.511333	-0.960433
1	0.573963	0.110074	-0.591796	0.688639
2	-0.675349	-0.476070	-0.109020	0.826062
3	-0.258913	-0.329534	-0.551565	0.826062
4	-0.675349	-0.476070	-0.551565	-0.905464

	PAYMENTS	MINIMUM_PAYMENTS	PRC_FULL_PAYMENT	TENURE	cluster
0	-0.528979	-3.109675e-01	-0.525551	0.36068	2
1	0.818642	8.931021e-02	0.234227	0.36068	0
2	-0.383805	-1.016632e-01	-0.525551	0.36068	1
3	-0.598688	-5.366135e-16	-0.525551	0.36068	2
4	-0.364368	-2.657913e-01	-0.525551	0.36068	2

Removing some redundant columns which seem unclear with the clustering seperation.

```
In [21]: data.columns
```

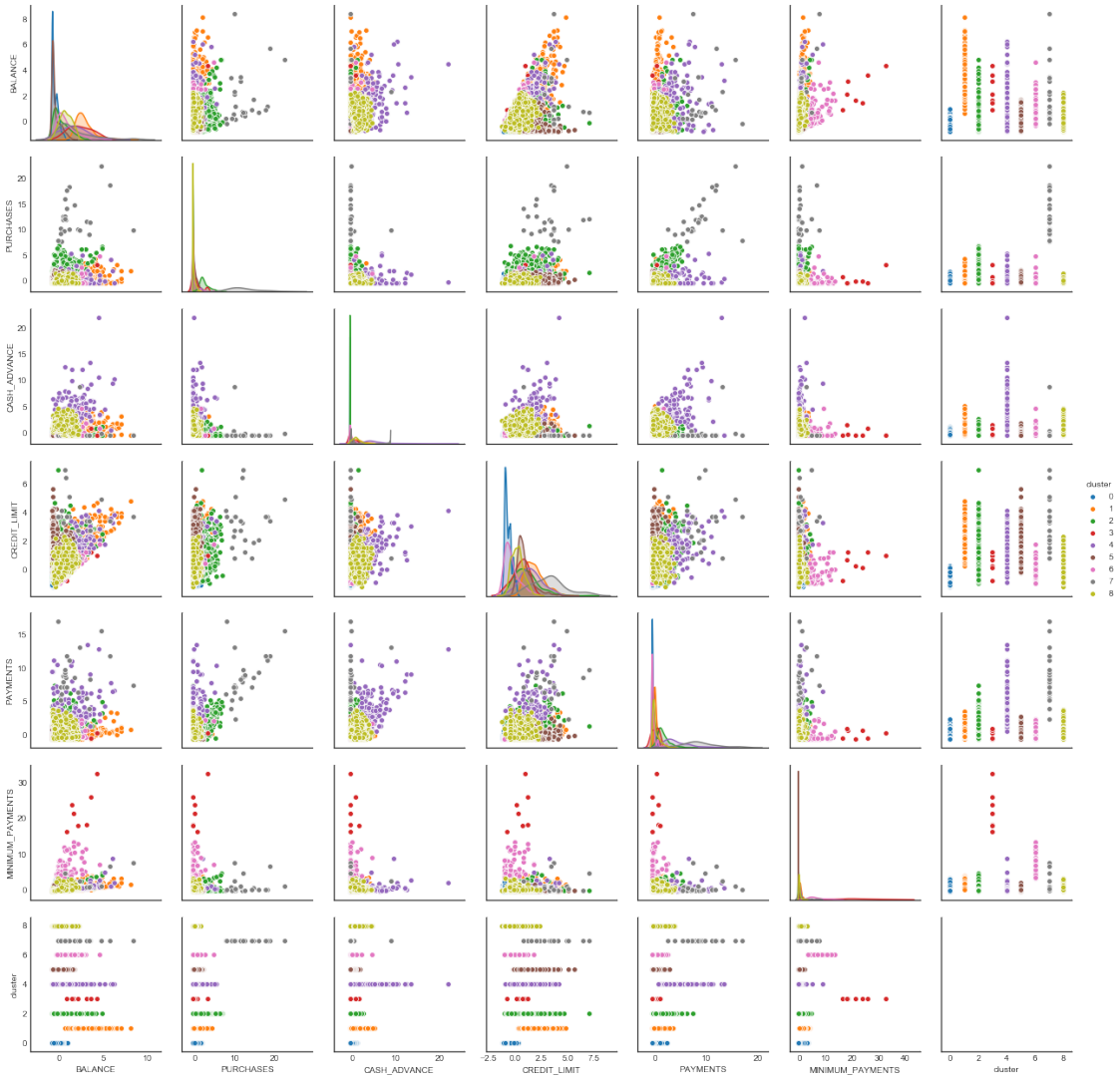
```
Out[21]: Index(['CUST_ID', 'BALANCE', 'BALANCE_FREQUENCY', 'PURCHASES',
                'ONEOFF_PURCHASES', 'INSTALLMENTS_PURCHASES', 'CASH_ADVANCE',
                'PURCHASES_FREQUENCY', 'ONEOFF_PURCHASES_FREQUENCY',
                'PURCHASES_INSTALLMENTS_FREQUENCY', 'CASH_ADVANCE_FREQUENCY',
                'CASH_ADVANCE_TRX', 'PURCHASES_TRX', 'CREDIT_LIMIT', 'PAYMENTS',
                'MINIMUM_PAYMENTS', 'PRC_FULL_PAYMENT', 'TENURE'],
               dtype='object')
```

Keep the following columns.

```
In [22]: col=['BALANCE','PURCHASES','CASH_ADVANCE','CREDIT_LIMIT','PAYMENTS','MINIMUM_PAYMENTS

In [23]: op_value=data_std[col].values

In [24]: op_value=pd.DataFrame(op_value)

In [25]: y_pred=kmeans.fit_predict(op_value)

In [26]: data_std["cluster"] = y_pred

In [27]: col.append('cluster')
```

In R, we used PCA to get a single plot for visualization. Here we plot we clusters for each pair of predictors to get a different view of our results.

```
In [28]: sns.pairplot(data_std[col],hue="cluster")

/Users/dingxuanzhang/anaconda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarn
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
/Users/dingxuanzhang/anaconda/lib/python3.6/site-packages/statsmodels/nonparametric/kde.py:488
  binned = fast_linbin(X, a, b, gridsize) / (delta * nobs)
/Users/dingxuanzhang/anaconda/lib/python3.6/site-packages/statsmodels/nonparametric/kdetools.py
  FAC1 = 2*(np.pi*bw/RANGE)**2
/Users/dingxuanzhang/anaconda/lib/python3.6/site-packages/numpy/core/fromnumeric.py:83: Runtime
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)


Out[28]: <seaborn.axisgrid.PairGrid at 0x1194260b8>
```



1.3 Analyzing the K-means method from `pairplot` graph

From the graph above, we can explore some potential market segments.

- Big consumers with large amount of payment. This group is observed from the second row intersecting with the fifth column.
- Small consumers with highest amount of minimal payment. This group is labeled as the pink color. This group shows the highest minimal payment, perhaps due to the second lowest limit. The banks maybe mark this group as the highest risk consumers.
- Cash Advances groups with high payment amount. This group takes more cash from the bank. Unlike the credit card consumption, they have to pay the money in advance.
- High credit limit but frugal This group has high credit limit, but it seems that they don't spend a lot on comsuming.