Boots UK - Significantly lifting incremental spend with tailored promotions for loyalty card customers

Business Challenge

Founded in 1849 and based in Nottingham, England, Boots UK is the UK's leading pharmacy, health and beauty retailer. Per the Company's website, Boots employs around 60,000 people and operates over 2,500 stores across the UK, excluding equity method investments, offering a range of services including eye and hearing care. Boots has more than 14 million active card members holders, of which around 90% are women.

Boots UK aims to inspire its ~15 million loyalty card holders to feel more engaged with the brand and increase their spend. How could it shape compelling offers based on unique customer preferences?

Business Solution

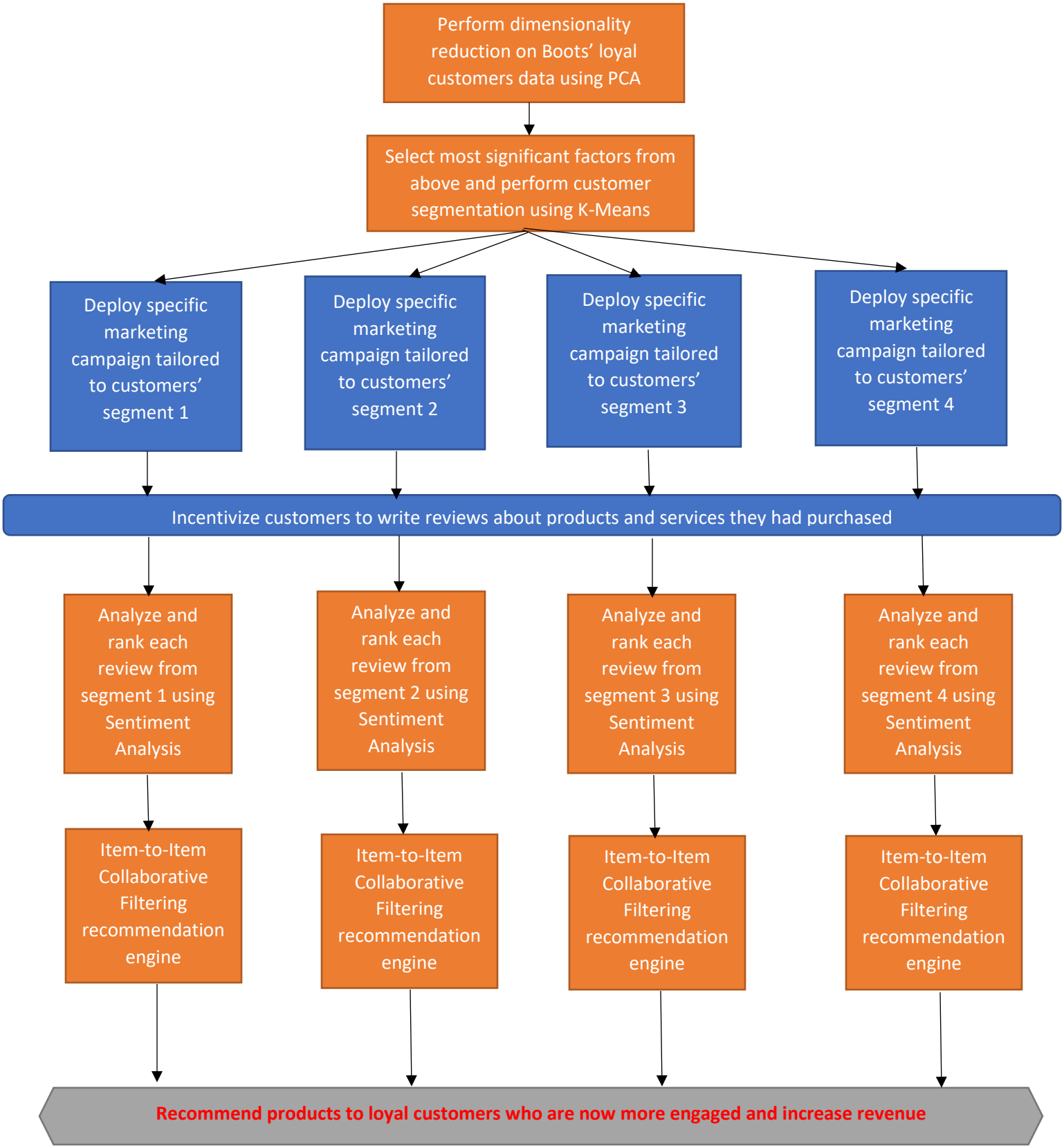
Today, Boots uses IBM solutions to match transactions to individual loyalty card customers and to determine the next-best actions for each person based on their unique purchasing histories and preferences. (see the success story here.) This document will suggest one approach to address the underlying business challenge. The solution suggested comprises of multiple steps, in which each step incorporates a machine learning model and/ or an analytic approach. **The business challenge is how we could help Boots increase its revenue derived from its loyal customers.** Since the customers are loyalty card holders, we assume Boots maintains large data set with information about those customers.

The first step of our business solution will be customer segmentation exercise using the K-Means algorithm so Boots could then deploy marketing campaign specifically tailored to each segment and to increase the effectiveness of marketing efforts. Note that since Boots' data about its loyal customers contains many features, we would have to perform Principal Component Analysis (PCA) to reduce dimensionality before we apply the K-Means clustering algorithm. We can view this approach as a tree-based model with our PCA/ K-Means output as the starting point and number of branches equals to the number of the K-Means output clusters.

Assuming customers react to the Boots' targeted advertisements effort and buy the suggested products and services, we will incentivize them to write reviews about the individual products and services they had purchased and post those reviews in their online loyal customer account under the "previous purchases" page that tracks each customer's past purchases. We will then create, using Natural Language Processing (NLP) algorithm of sentiment analysis, a system to rank their product satisfaction by giving a score based on their review. The final stage would be to predict and recommend customers about their next purchase using the Item-to-Item Collaborative Filtering approach.

Collaborative filtering can lead to some problems like Cold -Start for new items that are added to the list. With such new items or existing items with no reviews, the model will be less accurate because we do not know customers' preferences regarding those items. The term "Cold Start" derives from cars. When the engine is cold, the car is not yet working so smoothly, but once the optimal temperature is reached, it works perfect. For a recommendation engine it simply means that the conditions are not yet optimal for it to operate smoothly and provide best results. Basically, the reason we first perform the K-Means clustering to deploy tailored marketing campaigns (as described above) is to minimize the Cold-Start implications, because Boots has

~15 million loyal customers with thousands of products and no customers reviews about those products.



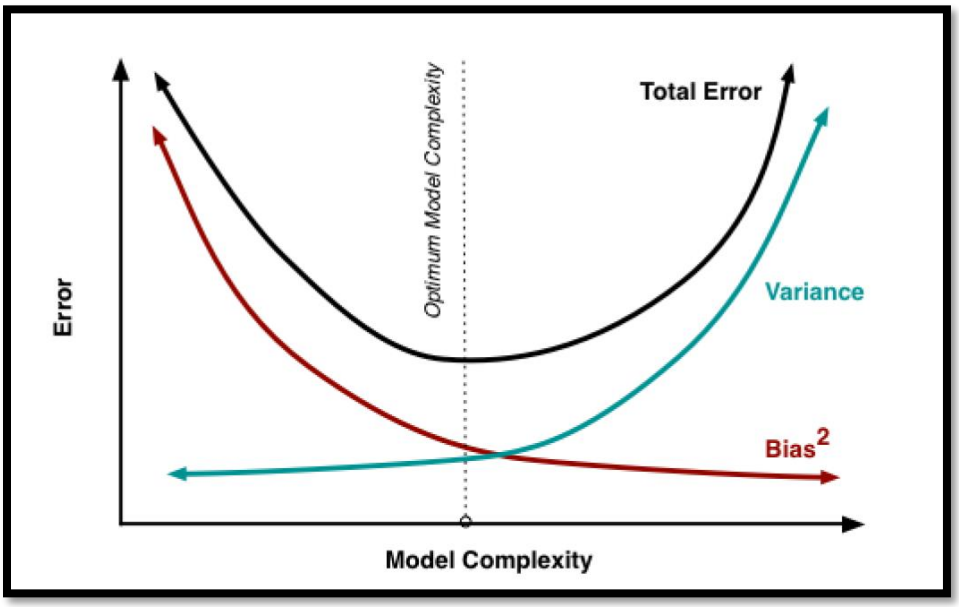
To prove the concept of the flow above and to deliver added value, I will include, together with descriptive analysis, some Python code to better explain the algorithms.
I will work with a "toy" customer data that I
retrieved from the Kaggle repository to proof the feasibility and logic behind my business solution (data can be found in this Kaggle page). The data contains "fake" data about customers with the following five features: Customer ID, Gender, Age, Annual Income and Spending Score (1-100). As mentioned above, Since Boots would like to increase incremental spending within the Company's loyalty card holders, we assume that the Company already maintains a data set about those customers. The *mall_customers* DataFrame taken from Kaggle (see below) is assumed to represent the data Boots maintains about its loyal customers.

Principal Component Analysis (PCA)

Bias is an error introduced in the model due to oversimplification of the algorithm and can lead to underfitting. However, variance is the error introduced in the model due to complex algorithm. As a result, the model is overfitting as it learns noise and randomness and will perform badly on test data set.

Principal component analysis (PCA) algorithm is used for dimensions reduction for the purpose of reducing the complexity of a model, to avoid overfitting and could be used as a step to balance the bias-variance tradeoff. (remember however that as we decrease the complexity of the model, we will see a reduction in error due to lower variance in the model. However, this only happens until a particular point. As we continue to make the model less complex, we could end up underfitting the model due to high bias. The goal is to have low bias and low variance to achieve good prediction performance which is what is called the bias-variance tradeoff.) In addition to the above, PCA is also used to reduce or eliminate correlation between variables, to simplify the model and to determine which of the variables account for the most variance.

Bias-Variance Trade-off:



The first step of our business solution is to apply PCA on the Boots' loyal customers data. As mentioned above, this data will be represented by the *mall_customers* DataFrame taken from Kaggle. The *mall_customers* (from here will be referred to as Boots' loyal customers data) data has only five features for simplicity, but obviously, the same PCA application can and will be performed on Boots' real data.

Let us apply a Principal Component Analysis on Boots' loyal customers data to reduce dimensionality. Boots' loyal customers data has five features, Customer ID, Gender, Age, Annual Income and Spending Score. Our PCA model will reduce the data from 5 dimensions to 2 (the 2 dimensions that explain ~60% of variability in the data. See below).

import libraries

In [48]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

Upload Boots' loyal customers data and observe first 5 observations.

In [71]:
```
mall_customers = pd.read_csv(r"C:\Users\...\Downloads\Mall_Customers.csv")
display (mall_customers.head())
```

	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

Observe the data's dimenstions and making sure we do not have missing data. Here we also rename fetures and conver the 'Gender' feature to a binary variable (Male == 1; Female == 0).

In [72]:
```
print ('\n', 'number of rows: {}, number of columns: {}'.format(mall_customers.shape[0], mall_customers.shape
[1]))
print ('\nchecking for missing data:\n', mall_customers.isnull().sum())
mall_customers['Annual Income (k$)'] = (mall_customers['Annual Income (k$)'] * 1000).astype(int)
# rename the col 'Annual Income (k$)'
mall_customers.rename(columns={'Annual Income (k$)': 'Annual Income'}, inplace=True)
# rename the col 'Genre'
mall_customers.rename(columns={'Genre': 'Gender'}, inplace=True)
# converting 'Male' to 1 and 'Female' to 0
mall_customers['Gender'] = np.where(mall_customers['Gender'] == 'Male', 1, 0)
 # drop 'CustomerID' col
mall_customers.drop(columns=['CustomerID'], inplace=True)
display (mall_customers.head())
```

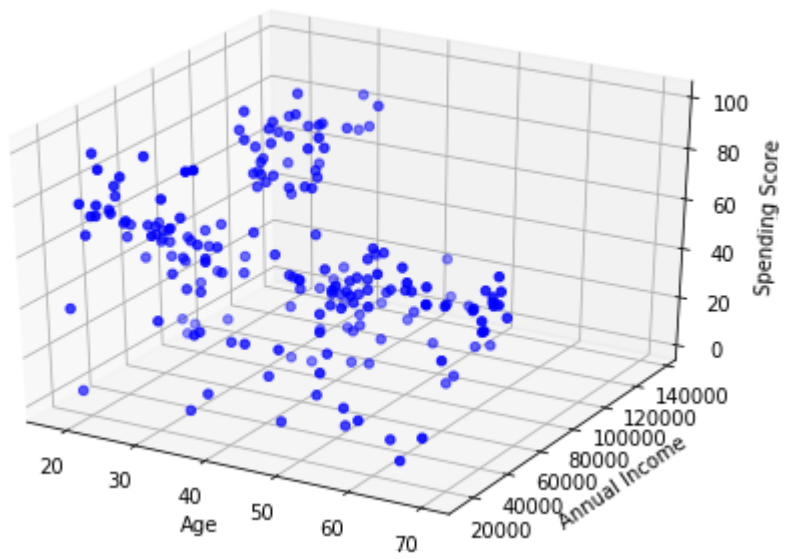
```
 number of rows: 200, number of columns: 5

checking for missing data:
 CustomerID                0
Genre                     0
Age                       0
Annual Income (k$)        0
Spending Score (1-100)    0
dtype: int64
```

	Gender	Age	Annual Income	Spending Score (1-100)
0	1	19	15000	39
1	1	21	15000	81
2	0	20	16000	6
3	0	23	16000	77
4	0	31	17000	40

Let us visualize Age, Annual Income and Spending Score in 3D space:

In [73]:
```
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(mall_customers['Age'], mall_customers['Annual Income'],
           mall_customers['Spending Score (1-100)'], c = 'b', marker='o')
ax.set_xlabel('Age')
ax.set_ylabel('Annual Income')
ax.set_zlabel('Spending Score')
print ('Let us visualize Age, Annual Income and Spending Score in 3D space:')
plt.show()
```

```
Let us visualize Age, Annual Income and Spending Score in 3D space:
```



Before we apply a Principal Component Analysis on Boots' data, let us explain the PCA algorithm and build one from scratch.

Let us assume an n X m matrix A such that each row i is an observation.

$A_{n,m}$ =

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{2,2} & \cdots & a_{n,m} \end{pmatrix}$$

Step 1 Compute mean of each row i (and store in a matrix $\overline{A}$ with n rows where all rows are identical and each contains the means). In other words, compute the mean of row 0, compute the mean of row 1, . . . compute the mean of row n to get a row matrix of length n and multiply it by an all-one-column matrix to get an n by m matrix $\overline{A}$ where each row is identical.

Step 2 Center matrix A in the space by subtracting $\overline{A}$ from A, such that B = $\overline{A}$ - A

Step 3 Compute the Covariance matrix C of B's rows (which is also $B^T B$).

Step 4 Compute the eigenvalues and eigenvectors of matrix C. Remember that given matrix X, eigenvectors and eigenvalues of matrix X are the vectors v's and values λ's such that $Xv = \lambda v$ for each v and λ.

The eigenvectors of our covariance matrix represent the vector directions of the new feature space and the eigenvalues represent the magnitudes of those vectors.

Step 5 Compute Principal Components such that PCs = $B\cdot$ eigenvectors.

Let us apply the above on Boots' loyal customer data (after scaling) and observe the PCs.

In [74]:
```
# Scale
A = StandardScaler().fit_transform(mall_customers)

Ave = np.mean(A.T)
B = A - Ave
C = np.cov(B.T)
values, vectors = np.linalg.eig(C)
P = B.dot(vectors)
PCs = pd.DataFrame(data = P, columns = ['PC0', 'PC1', 'PC2', 'PC3'])
print(PCs)
```

```
          PC0       PC1       PC2       PC3
0   -0.406383 -0.520714  2.072527 -1.335529
1   -1.427673 -0.367310  2.277644 -0.082329
2    0.050761 -1.894068  0.367375 -2.174381
3   -1.694513 -1.631908  0.717467 -0.075228
4   -0.313108 -1.810483  0.426460 -0.683070
..        ...       ...       ...       ...
195 -1.179572  1.324568 -1.932441  0.615899
196  0.672751  1.221061 -2.438084 -0.272925
197 -0.723719  2.765010 -0.583178  0.313022
198  0.767096  2.861930 -1.150341 -1.219621
199 -1.065015  3.137256 -0.788146  0.461014

[200 rows x 4 columns]
```

Observe the PCs results above, calculated manually and the PCs below calculated using the sklearn library. We got the same PCs other than PC2 and PC3's signs which are reversed. However, note that the PCA sign does not affect its interpretation since the sign does not affect the variance contained in each component. Only the relative signs of features forming the PCA dimension are important. See further discussion following this link: (https://stackoverflow.com/questions/22984335/recovering-features-names-of-explained-variance-ratio-in-pca-with-sklearn). Hence, we can conclude that the manual computation above is correct.

In [75]:
```
pca = PCA(n_components=4)
pca_fit = pca.fit_transform(A)
pca_df = pd.DataFrame(data = pca_fit
             , columns = ['PC0', 'PC1', 'PC2', 'PC3'])


print(pca_df)
```

```
          PC0       PC1       PC2       PC3
0   -0.406383 -0.520714 -2.072527  1.335529
1   -1.427673 -0.367310 -2.277644  0.082329
2    0.050761 -1.894068 -0.367375  2.174381
3   -1.694513 -1.631908 -0.717467  0.075228
4   -0.313108 -1.810483 -0.426460  0.683070
..        ...       ...       ...       ...
195 -1.179572  1.324568  1.932441 -0.615899
196  0.672751  1.221061  2.438084  0.272925
197 -0.723719  2.765010  0.583178 -0.313022
198  0.767096  2.861930  1.150341  1.219621
199 -1.065015  3.137256  0.788146 -0.461014

[200 rows x 4 columns]
```

Observe the percentage of explained variance of each PC.

In [59]:
```
print (pca.explained_variance_ratio_)
```

```
[0.33690046 0.26230645 0.23260639 0.16818671]
```

As mentioned above, we would like to reduce Boots' loyal cutomers data from 5 dimensions to 2 (which will explain ~60% of variance).

In [60]:
```
pca = PCA(n_components=2)
pca_fit = pca.fit_transform(A)
pca_df = pd.DataFrame(data = pca_fit
             , columns = ['PC0', 'PC1'])

print(pca_df)
```

```
          PC0       PC1
0   -0.406383 -0.520714
1   -1.427673 -0.367310
2    0.050761 -1.894068
3   -1.694513 -1.631908
4   -0.313108 -1.810483
..        ...       ...
195 -1.179572  1.324568
196  0.672751  1.221061
197 -0.723719  2.765010
198  0.767096  2.861930
199 -1.065015  3.137256

[200 rows x 2 columns]
```

Now, that we have our two PCs, which features do they really represent?

Note: Many people (including myself at the beginning) think that the columns above PC0 and PC1 represent the "best" two columns in the original data. This is not correct!!! What it really means is that the PC0 column is informed by the Age column more than any other column and PC1 by Annual Income (see below). It does not mean that the PC0 column is one for one the Age column and PC1 is the Annual Income column. I really recommend to carefully read through these posts (https://stackoverflow.com/questions/22984335/recovering-features-names-of-explained-variance-ratio-in-pca-with-sklearn) to really understand the PCA algorithm.

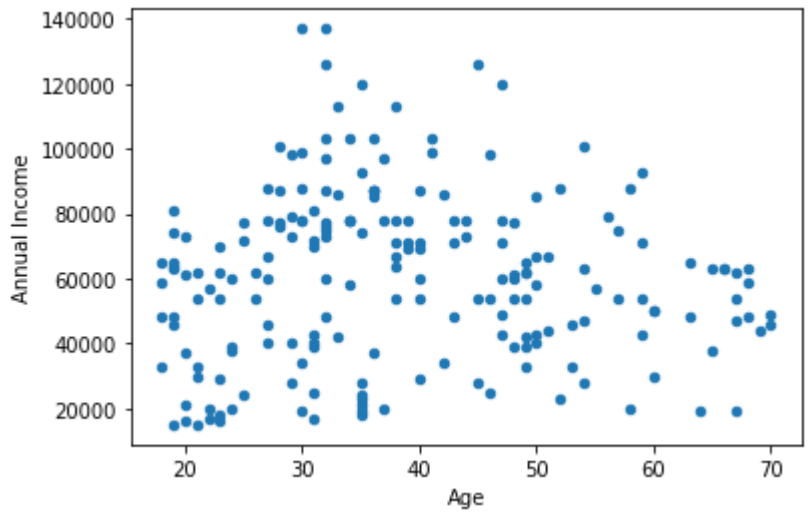
In [44]:
```
most_important = [np.abs(pca.components_[i]).argmax() for i in range(2)]
initial_feature_names = ['Gender', 'Age', 'Annual Income', 'Spending Score (1-100)']
most_important_names = [initial_feature_names[most_important[i]] for i in range(2)]
dic = {'PC{}'.format(i): most_important_names[i] for i in range(2)}
df = pd.DataFrame(dic.items())
print (df)
```

```
     0              1
0  PC0            Age
1  PC1  Annual Income
```

Plot the 'Age' and 'Annual Income' features.

In [10]:
```
df_final = mall_customers[['Age','Annual Income']]
df_final.plot.scatter(x='Age', y='Annual Income')
```

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x29601dc3248>



Now that we reduced Boots' loyal customers data to 2 most important dimensions/ features, let us use the K-Means algorithm to cluster the customers into distinct segments based on this data so we can deploy targeted marketing initiatives and offer each segment specific and tailored products and services.

But before we do that, let us try to undertand the K-Means algorithm.

K-Means Clustering

Our goal in clustering is to group similar instances, or observations into clusters. Clustering techniques are commonly used in customers segmentations (to develop marketing strategies or to recommend products), recommender systems used by Netflix, Amazon, retailers, etc., search engines, image segmentations (for objects detection and tracking systems), anomalies detection (in production or fraud detection), and more. K-Means is an unsupervised algorithm, because we are trying to label unlabeled data. To better understand the K-means algorithm we will attempt to write the algorithm from scratch before we actually apply it to the PCA output from above. For the purpose of K-Means, we will start with easy-to-visualize "toy" data retrieved from GitHub and can be found here. This data was purposely selected, because it will be easy to understand and visualize the K-Means algorithm using this data. Observe that when we plot the observations, we can easily see that the data should be clustered into 3 unique clusters.

import libraries

In [1]:
```
import pandas as pd
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from scipy.spatial import distance
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.metrics import accuracy_score
```

Upload the data and observe first 5 observations

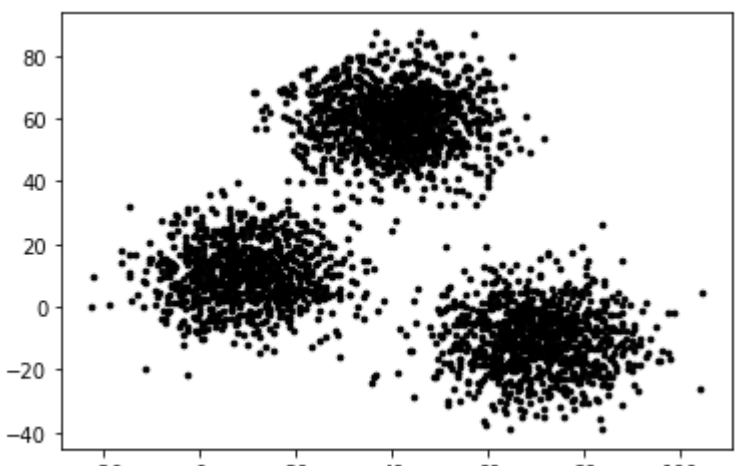
In [3]:
```
df = pd.read_csv("https://raw.githubusercontent.com/mubaris/friendly-fortnight/master/xclara.csv")
display (df.head())

# A visualization plot function to be used later
def plot(x, y, c):
  fig, ax = plt.subplots()
  scatter = ax.scatter(x, y, c=c, cmap=plt.cm.brg)
  legend = ax.legend(*scatter.legend_elements(),
                     loc="upper right", title="labels")
  ax.add_artist(legend)
  ax.set_title('K-Means Clusters')
  plt.show()
```

	V1	V2
0	2.072345	-3.241693
1	17.936710	15.784810
2	1.083576	7.319176
3	11.120670	14.406780
4	23.711550	2.557729

Observe the data plot. For simplicity and better understanding of the K-Means algorithm we will use this data as it is easy separable into 3 clusters.

In [4]:
```
X = df.to_numpy()
plt.scatter(X[:, 0], X[:, 1], c='black', s=7)
```

Out[4]: `<matplotlib.collections.PathCollection at 0x1edc1e5cb48>`



The K-Means Algorithm (from scratch)

Let us assume that X is a 2D NumPy array, or a matrix such that each row represent an observation. Also, let us define the number of clusters to be k. In our example, we can clearly see that k == 3.

We will follow the following steps:

1. Randomly samples k observations from X, ***with no replacement***, to be used as initial centers (centroids). Note that k is predefined (more on this later).
2. Compute the squared Euclidean distance from each point in matrix X to each centroid (to calculate the Within-Cluster Sum of Squares - WCSS; inertia per sklearn documentation; distortion per Scipy documentation) .
3. Assign each data point to the closest centroid (cluster).
4. Update the centroids by computing the mean of each cluster.
5. Iterate steps 2, 3 and 4 until the same points are assigned to each cluster in consecutive rounds.

The objective of K-Means clustering is to minimize total intra-cluster variance or the WCSS.

To visualize the above (slightly different but mainly the same) , click [here (http://shabal.in/visuals/kmeans/5.html)](http://shabal.in/visuals/kmeans/5.html)

Step 1: Let us write a function, *initial_centers(X, k)*, such that it randomly selects k observations from matrix X, which is a NumPy array of size m by d, as centroids. It should return a NumPy array of size k by d.

In [15]:
```
def initial_centers(X, k):
    samples = np.random.choice(len(X), size=k, replace=False)
    return X[samples, :]

k=3
centers_initial = initial_centers(X, k)
print("Initial centers:\n", centers_initial)
```

```
Initial centers:
 [[19.2699   67.18515 ]
 [ 3.509066 16.55414 ]
 [-7.409304 15.4642  ]]
```

Step 2: Let us write a function, *d2(X, centers)*, that computes the squared Euclidean distance from each point to each centroid, such that $S_{ij} = d_{ij}^2$ and d_{ij}^2 is the squared distance from point $\hat{x}_i$ to center $\hat{\mu}_j$. It should return a NumPy array $S[:m, :k]$.

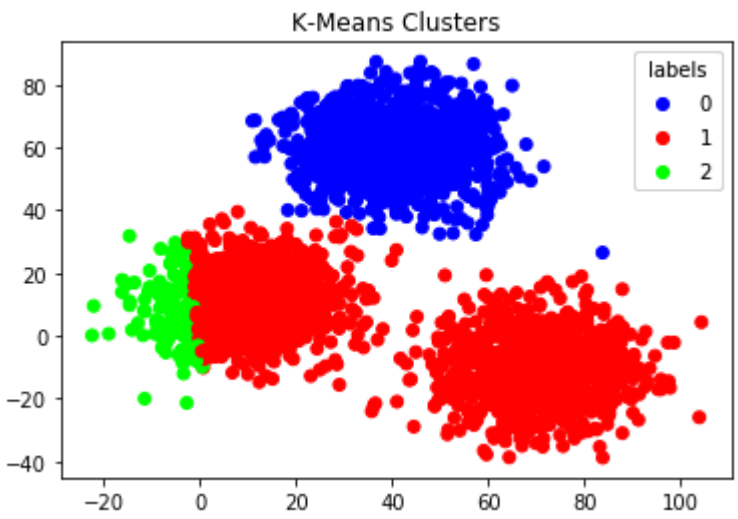
In [16]:
```
def d2(X, centers):
    S = np.zeros((len(X), len(centers)))
    S = sp.spatial.distance.cdist(X, centers, metric='euclidean')**2
    return S

S = d2(X, centers_initial)
print ('k = ', k)
print ('X shape: ',X.shape)
print ('S shape: ', S.shape)
```

```
k =  3
X shape:  (3000, 2)
S shape:  (3000, 3)
```

Step 3: Let us write a function, *assign_cluster_labels(S)*, that assign a "cluster label" to each point. In particular, consider the m by k squared distance matrix S_{ij}, for each point $\hat{x}_i$, the cluster label is the index j^{th} minimum squared distance. It should return a column vector y of length m such that $y_i = argmin_{j \in = \{0, \ldots k-1\}} S_{ij}$

Let us also visualize the initial clusters, as assigned by the initial centroids to demonstrate, as expected, the "poor" initial clusters.

In [17]:
```
def assign_cluster_labels(S):
    y = np.argmin(S, axis=1)
    return y

y = assign_cluster_labels(S)
df['labels'] = y
plot(df['V1'], df['V2'], df['labels'])
```



Step 4: Let us write a function, *update_centers(X, y)*, that given a clustering (a set of points and assignment of labels), computes the center of each cluster. Observe that by calculating the mean of a cluster we basically updating its' centroid. It should return a NumPy array of size k by d

In [18]:
```
def update_centers(X, y):
    m, d = X.shape
    k = int(max(y) + 1)
    centers = np.zeros((k, d))

    for i in range(k):
        centers[i,:] = np.mean(X[y == i, :], axis = 0)
    return centers

new_centers = update_centers(X, y)
print ('New centers: \n', new_centers)
```

```
New centers:
 [[40.74693199 59.77968829]
 [44.06298891 -0.53083259]
 [-5.97281431  7.58415924]]
```

Step 5: Let us write a function, *WCSS(S)*, that given the squared distances, S, returns the within-cluster sum of squares.

for example, suppose S is defined as:

$$\begin{bmatrix} 3 & 2 & 7 \\ 1 & 5 & 3 \\ 4 & 2 & 5 \end{bmatrix}$$

Then the WCSS == 2 + 1 + 2 == 5

In [19]:
```
def WCSS(S):
    mini = sum(np.amin(S, axis=1))
    return mini
print (WCSS(S))
```

```
6126735.431440922
```

In [20]:
```
# To be used later, a function to check whether the centers have "moved," given two instances of the center v
alues.
# It accounts for the fact that the order of centers may have changed (set).

def inertia(old_centers, centers):
    return set([tuple(x) for x in old_centers]) == set([tuple(x) for x in centers])
```

Put all of the preceding building blocks together to implement Lloyd's K -Means algorithm.

Observe below how the WCSS is reduced with each iteration.

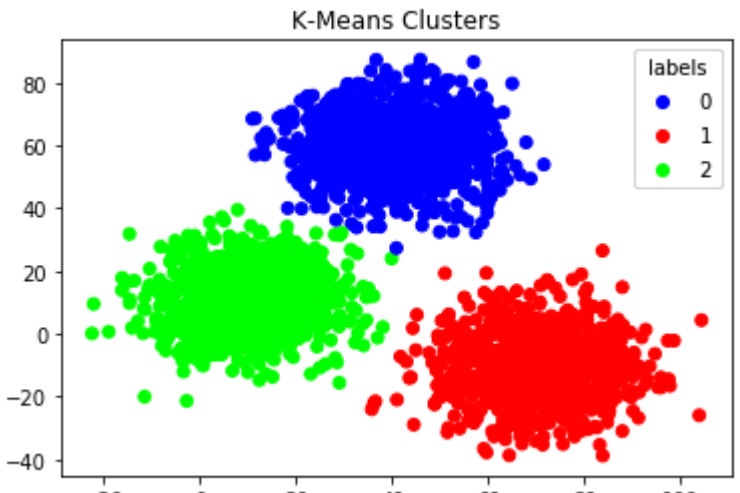
In [21]:
```
def kmeans(X, k):
    centers = initial_centers(X, k)
    labels = np.zeros(len(X))
    i = 1
    converged = False
    while (not converged):
# Random centers
        old_centers = centers
# Compute distance^2 matrix
        S = d2(X, old_centers)
# Assign the points to a cluster
        labels = assign_cluster_labels(S)
# Recalculate the centroids
        centers = update_centers(X, labels)
        converged = inertia(old_centers, centers)
        print ("iteration", i, "WCSS = ", WCSS (S))
        i += 1
    return labels

clustering = kmeans(X, k)
```

```
iteration 1 WCSS =  2940217.996929049
iteration 2 WCSS =  1789390.3791812463
iteration 3 WCSS =  729712.3486699627
iteration 4 WCSS =  611803.4214986623
iteration 5 WCSS =  611605.8806933895
```

Observe below a perfect clustering.

In [22]:
```
df['labels'] = clustering
centers = update_centers(X, clustering)
display (df.head())

plot(df['V1'], df['V2'], df['labels'])
```

	V1	V2	labels
0	2.072345	-3.241693	2
1	17.936710	15.784810	2
2	1.083576	7.319176	2
3	11.120670	14.406780	2
4	23.711550	2.557729	2



It was easy to see that our "synthetic" data was calling for 3 clusters, but in reality it is usually not that obvious. So, how can we determine the number of clusters, or k value.

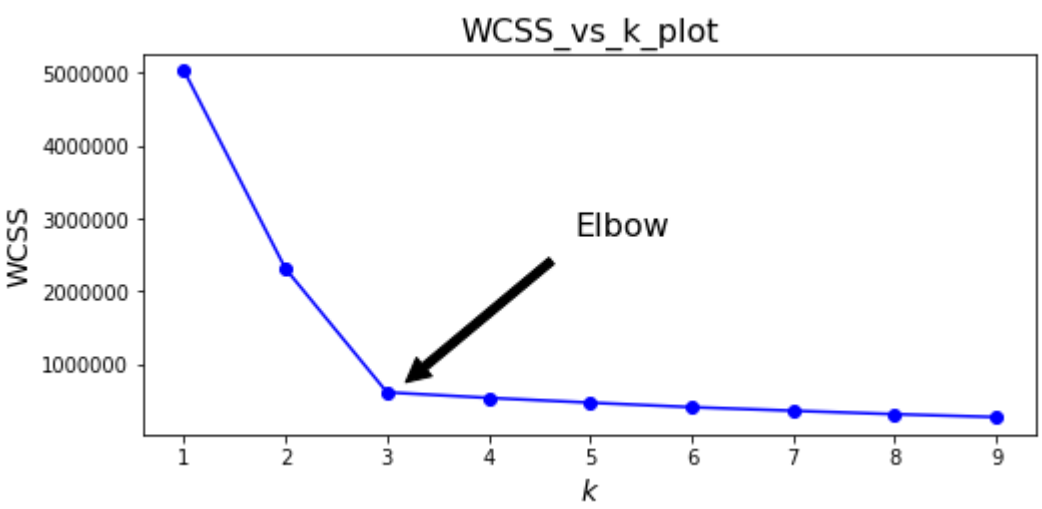
The objective of K-Means clustering is to minimize total WCSS. So one might argue, why not have k == number of observations with WCSS == 0 (if number of clusters equals number of observations WCSS would be 0, since each point is a centroid.). However, doing so defeat the purpose of the algorithm.

There are different methods to determine the optimal number of clusters. Let us see two of those methods, the "Elbow" and Silhouette Coefficient methods.

The "Elbow" Method

The elbow method plots the value of the cost function produced by different values of k. As we know, if k increases, average WCSS will decrease, each cluster will have fewer constituent instances, and the instances will be closer to their respective centroids. However, the improvements in average WCSS will decline as k increases. The value of k at which improvement in WCSS declines the most is called the elbow, at which we should stop dividing the data into further clusters.

In [23]:
```
kmeans_per_k = [KMeans(n_clusters=k, random_state=0).fit(X)   # explained below
                for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]   # explained below

plt.figure(figsize=(8, 3.5))
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("WCSS", fontsize=14)
plt.annotate('Elbow',
             xy=(3, inertias[3]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=16,
             arrowprops=dict(facecolor='black', shrink=0.1))

plt.title("WCSS_vs_k_plot", fontsize=16)
plt.show()
```



The Silhouette Coefficient Method

The silhouette coefficient is a measure of the compactness and separation of the clusters. Higher values represent a better quality of cluster. The best value is 1 and the worst value is -1. Values near 0 indicate overlapping clusters. Negative values generally indicate that a sample has been assigned to the wrong cluster, as a different cluster is more similar.

The Silhouette Coefficient is calculated using the mean intra-cluster distance (a) and the mean nearest-cluster distance (b) for each sample. The Silhouette Coefficient for a sample is:

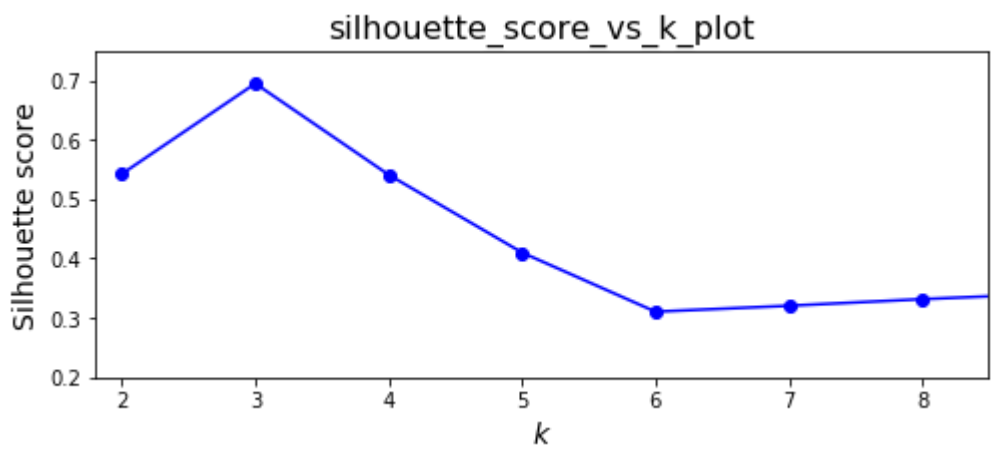
$$\frac{b-a}{max(a,b)}$$

a is the mean distance between the instances in the cluster, b is the mean distance between the instance and the instances in the next closest cluster.

(Click here (https://www.youtube.com/watch?v=_j37uExzbXk) for a very short video explaining the silhouette coefficient method)

In [24]:
```
kmeans = KMeans(n_clusters=k)   # explained below
clusters = kmeans.fit_predict(X)   # explained below
print ('silhouette score: ', silhouette_score(X, kmeans.labels_))
silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]
plt.figure(figsize=(8, 3))
plt.plot(range(2, 10), silhouette_scores, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Silhouette score", fontsize=14)
plt.axis([1.8, 8.5, 0.2, 0.75])
plt.title("silhouette_score_vs_k_plot", fontsize=16)
plt.show()
```

```
silhouette score:  0.6945587736089913
```

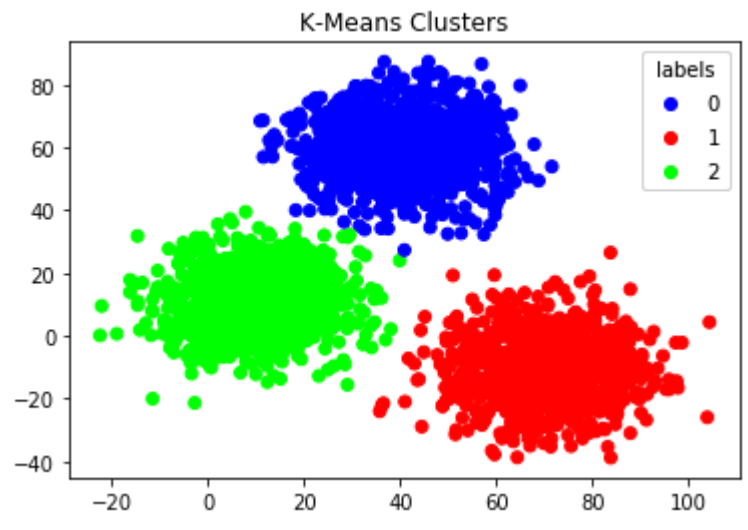


K-Means Clustering with sklearn

Now that we built the K-Means algorithm from scratch and understand its goal and function, let's see how simple it is to do using the sklearn.cluster.KMeans library.

In [25]:
```
# reload the "toy" data as Pandas DataFrame
km_df = pd.read_csv("https://raw.githubusercontent.com/mubaris/friendly-fortnight/master/xclara.csv")

# K-means using sklearn
kmeans = KMeans(n_clusters=k)
clusters = kmeans.fit_predict(X)
#---------------------------------------

km_df['labels'] = kmeans.labels_
WCSS = kmeans.inertia_
centroids = kmeans.cluster_centers_

print ('WCSS: {}'.format(WCSS), '\nCentroids:\n {}'.format(centroids))
plot(km_df['V1'], km_df['V2'], km_df['labels'])
```

```
WCSS: 611605.8806933891
Centroids:
 [[ 40.68362784  59.71589274]
 [ 69.92418447 -10.11964119]
 [  9.4780459   10.686052  ]]
```



So with the K-Means understanding and sklearn's KMeans library, let us perform the customer segmentation task by utilizing the K-Means algorithm on Boots' loyal customers data's "most important" Principal Components from above so Boots' marketing department could deploy their targeted marketing and segment-based campaigns.

Reload Boots' loyal customers data.

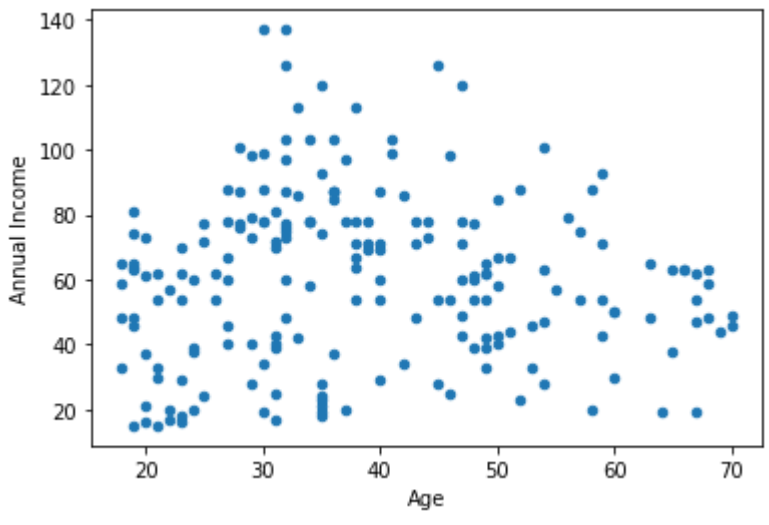
In [28]:
```
mall_customers = pd.read_csv(r"C:\Users\...\Downloads\Mall_Customers.csv")
mall_customers.rename(columns={'Annual Income (k$)': 'Annual Income'}, inplace=True)
display (mall_customers.head())
```

	CustomerID	Genre	Age	Annual Income	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

Remeber that from the PCA performed above we determined to include only 'Age' and 'Annual Income.'

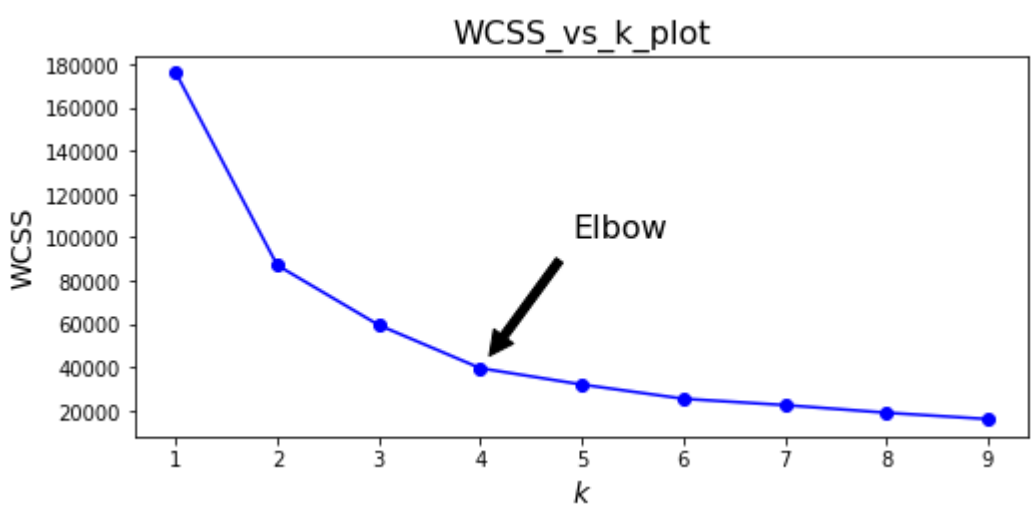
In [29]:
```
df_final = mall_customers[['Age','Annual Income']]
df_final.plot.scatter(x='Age', y='Annual Income')
```

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x1edc5b415c8>



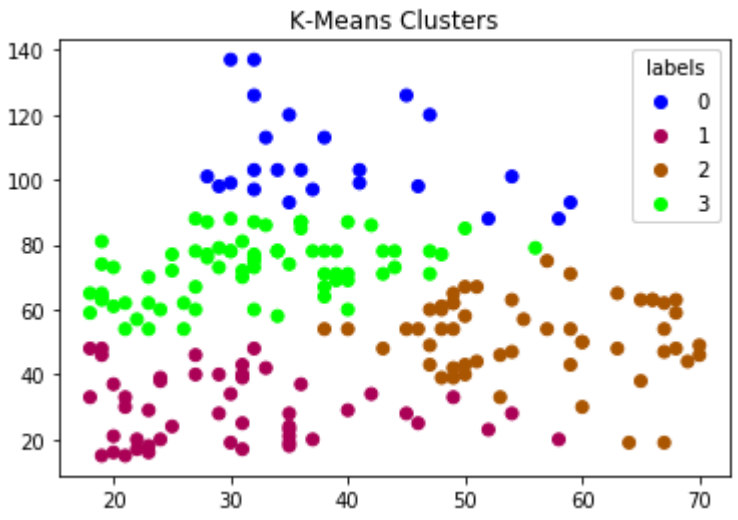
Perform Elbow method. Observe below I determined the optimal k == 4.

In [45]:
```
X = df_final
k=10
# K-means using sklearn
kmeans = KMeans(n_clusters=k)
clusters = kmeans.fit_predict(X)
#----------------------------------------
kmeans_per_k = [KMeans(n_clusters=k, random_state=0).fit(X)
                for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]

# Elbow
plt.figure(figsize=(8, 3.5))
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("WCSS", fontsize=14)
plt.title("WCSS_vs_k_plot", fontsize=16)
plt.annotate('Elbow',
             xy=(4, inertias[3]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=16,
             arrowprops=dict(facecolor='black', shrink=0.1))
plt.show()
```



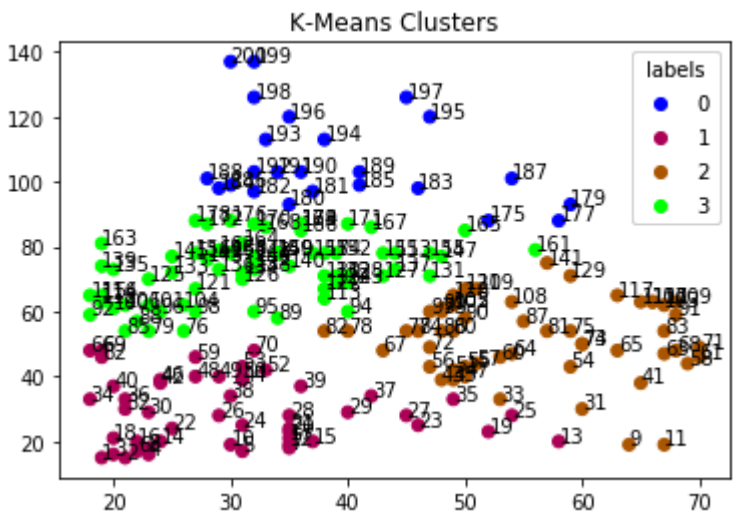
In [49]:
```
# k == 4
kmeans = KMeans(n_clusters=4)
clusters = kmeans.fit_predict(X)

df_final['labels'] = kmeans.labels_
WCSS = kmeans.inertia_
centroids = kmeans.cluster_centers_

def plot(x, y, c):
    fig, ax = plt.subplots()
    scatter = ax.scatter(x, y, c=c, cmap=plt.cm.brg)
    legend = ax.legend(*scatter.legend_elements(),
                       loc="upper right", title="labels")
    ax.add_artist(legend)
    ax.set_title('K-Means Clusters')
    plt.show()

print ('WCSS: {}'.format(WCSS), '\nCentroids:\n {}'.format(centroids))
plot(df_final['Age'], df_final['Annual Income'], df_final['labels'])
```

```
WCSS: 39537.56143675126
Centroids:
 [[ 39.         106.5          3.        ]
 [ 30.34693878  29.26530612   1.91836735]
 [ 55.81481481  51.77777778   0.        ]
 [ 31.95890411  72.95890411   1.        ]]
```



Let us also add the Customer ID to each point so we know to which segment each customer belongs to.

In [51]:
```
ID = mall_customers['CustomerID'].tolist()
def plot(x, y, c):
    fig, ax = plt.subplots()
    scatter = ax.scatter(x, y, c=c, cmap=plt.cm.brg)
    legend = ax.legend(*scatter.legend_elements(),
                       loc="upper right", title="labels")
    ax.add_artist(legend)
    ax.set_title('K-Means Clusters')
    for i, id in enumerate(ID):
        ax.annotate(id, (x[i], y[i]))
    plt.show()

plot(df_final['Age'], df_final['Annual Income'], df_final['labels'])
```



Item-to-Item Collaborative Filtering

Recommendation algorithms provide an effective form of targeted marketing by creating a personalized shopping experience for each customer. Collaborative Filtering is the most common technique used when it comes to building intelligent recommender systems that can learn to give better recommendations. Collaborative filtering works by finding out similarities between two users (user-to-user collaborative filtering) or two items (item-to-item collaborative filtering). More specifically:

> **User-based**: For a user U, with a set of similar users determined based on rating vectors consisting of given item ratings, the rating for an item I, which hasn't been rated, is found by picking out N users from the similarity list who have rated the item I and calculating the rating based on these N ratings.
>
> **Item-based**: For an item I, with a set of similar items determined based on rating vectors consisting of received user ratings, the rating by a user U, who hasn't rated it, is found by picking out N items from the similarity list that have been rated by U and calculating the rating based on these N ratings.

Simply put, Item-to-Item Collaborative Filtering matches each of the customer's purchased and rated items to similar items, then combines those similar items into a recommendation list from items not previously purchased by the customer.

As mentioned above, after segmenting the entire population of customers into different segments (using the K-Means algorithm), deploying targeted marketing campaigns, incentivizing customers to write reviews in their online loyal customer page about products suggested by these campaigns, we would need to rank each review and score it. To do this we will use the Sentiment Analysis algorithm which is a model within the broad application of Natural Language Processing (NLP). Note that I will not explain the Sentiment Analysis algorithm here, because this by it self can be a separate project. I will however go through some real examples to show how it ranks reviews. In general, Sentiment Analysis refers to the use of NLP, text analysis and computational linguistics to determine subjective information or the emotional state of the writer. It is commonly used in reviews which save businesses a lot of time from manually reading comments. The library used below reads the review and returns a rank between -1 and 1, where -1 means a bad review (the customer did not like the product) and 1 means a good review (the customer loved the product). Let us see some real examples in action below.

In [ ]:
```
import nltk
nltk.download('vader_lexicon')
```

In [1]:
```
from nltk.sentiment.vader import SentimentIntensityAnalyzer
sia = SentimentIntensityAnalyzer()
```

Original 5 stars Amazon review.

I wrote this review about a backpack (https://www.amazon.com/gp/product/B07RYLCTJJ/ref=ppx_yo_dt_b_asin_title_o00_s00?ie=UTF8&psc=1) I purchased on Amazon.

In [2]:
```
review0 = '''An awesome backpack that you can walk around with all day and won't feel a thing.
      I love the fact that I can reach the baby-wipes dispenser and three bottles without taking the bag off.
      I have two kids (2 and half and 8 months) and I could fit in the bag everything I needed for the entire
day,
      including food in the insulated pocket. Overall great purchase, worth every penny. Highly recommended!
      '''
print("Sentiment Score: ", sia.polarity_scores(review0)['compound'])
```

```
Sentiment Score:  0.8938
```

Original 1 star Amazon review.

I worte this review about this book (https://www.amazon.com/The-Deep-Learning-Revolution/dp/B07MM8F42R/ref=sr_1_1?crid=1W1U8BBPO30QZ&dchild=1&keywords=the+deep+learning+revolution&qid=1585931282&s=baby-products&sprefix=the+deep+lear%2Cbaby-products%2C192&sr=8-1) I purchased on Amazon. (DO NOT BUY THIS BOOK!)

In [3]:
```
review1 = '''
          Did not like the book and stopped reading half into it. The book is not for an average reader.
          Felt like you would need a PhD in neuroscience to understand it. Waste of money!
          '''
print("Sentiment Score: ", sia.polarity_scores(review1)['compound'])
```

```
Sentiment Score:  -0.8027
```

Original 5 stars Amazon review.

This is a real Amazon review about this book (https://www.amazon.com/gp/product/1492032646/ref=ppx_yo_dt_b_asin_title_o05_s01?ie=UTF8&psc=1). (I own this book and can also recommend it)

In [4]:
```
review2 = '''
          Aurelien did it again!
          Whether you are a data scientist looking to start building predictive models in Python,
          or a software developer looking to become an ML engineer, look no further!
          The excellent balance between theory/background and implementation that was present in the first ed
ition is kept,
          with the essential material additions made (e.g. the unsupervised learning in the "classical ML" pa
rt,
          or the Keras API, which is quickly becoming the most popular way to use TensorFlow).
          Needless to say, the Jupyter notes accompanying each chapter are more than helpful.
          Also, as a cherry on top, the illustrations in the printed version are now in color,
          which makes it even easier to read. In summary, this book is an absolute must-have for a
          Python-rooted data scientist / ML engineer!
          '''
print("Sentiment Score: ", sia.polarity_scores(review2)['compound'])
```

```
Sentiment Score:  0.9209
```

Original 3 stars Amazon review.

A 3 stars Amazon review given to the book (https://www.amazon.com/gp/product/1492032646/ref=ppx_yo_dt_b_asin_title_o05_s01?ie=UTF8&psc=1) above.

In [5]:
```
review3 = '''
          Its quality is ok, but I think it is too expensive
          '''
print("Sentiment Score: ", sia.polarity_scores(review3)['compound'])
```

```
Sentiment Score:  0.1531
```

Working with Boots' loyal customers reviews we collect, we can rank Sentiment Analysis' Score between -1 and -0.6 as 1 star review, -0.61 and -0.2 as 2 stars review, -0.21 and 0.2 as 3 stars review, 0.21 and 0.6 as 4 stars review, and 0.61 and 1 as 5 stars review.

Now that we have the Sentiment Analysis system in place and we are starting to get customers' reviews, let us build the Item-to-Item Collaborative Filtering model.

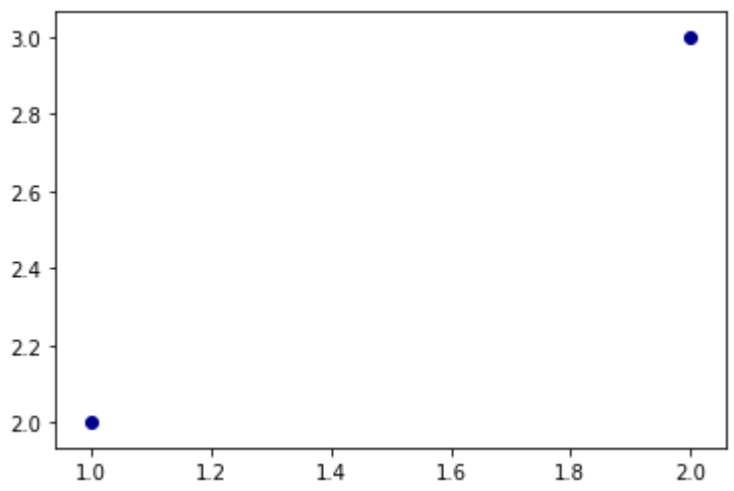
Remember that with item-to-item CF we are trying to find similar products. So the question is how can we measure this similarity between products? To answer this question we will start with the products reviews stars ranking system we developed above. Let us plot 2 dummy points of Item 1 and Item 2 with ranks of 1 and 2 for Item 1 (from two different Users), and 2 and 3 for Item 2 (from the same two Users) to see how can we measure Items similarity. We can see that Item 1 and Item 2 are clearly ***review-based*** similar products. ***Both products pretty much SUCK!!!***

In [6]:
```
import pandas as pd
import numpy as np
from scipy import spatial
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

In [7]:
```
I1 = np.array([1,2])
I2 = np.array([2,3])
sim = pd.DataFrame([I1, I2], columns=['Item1', 'Item2'])
sim = sim.set_index([pd.Series(['User1', 'User2'])])
display(sim)

plt.scatter(x = [1,2], y = [2,3], c='DarkBlue')
```

	Item1	Item2
User1	1	2
User2	2	3

Out[7]: <matplotlib.collections.PathCollection at 0x179e8ad9248>



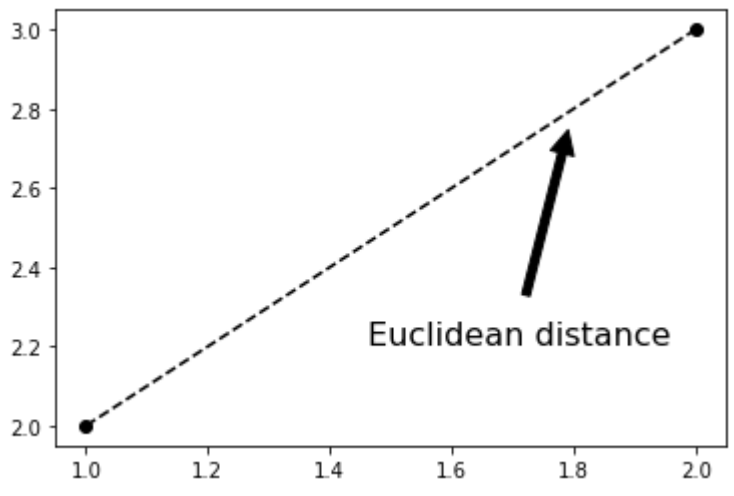
So how to calculate product similarity? Looking at the Euclidean distance between the points seems to be a good way to estimate similarity, right? ***Wrong!*** A common mistake would be to calculate the Euclidean distance between the two points. Let us compute the Euclidean distance and check the result.

In [8]:
```
I1 = [1,2]
I2 = [2,3]
Euc = spatial.distance.euclidean(I1, I2)
print ("The Euclidean distance is: ", Euc)
```

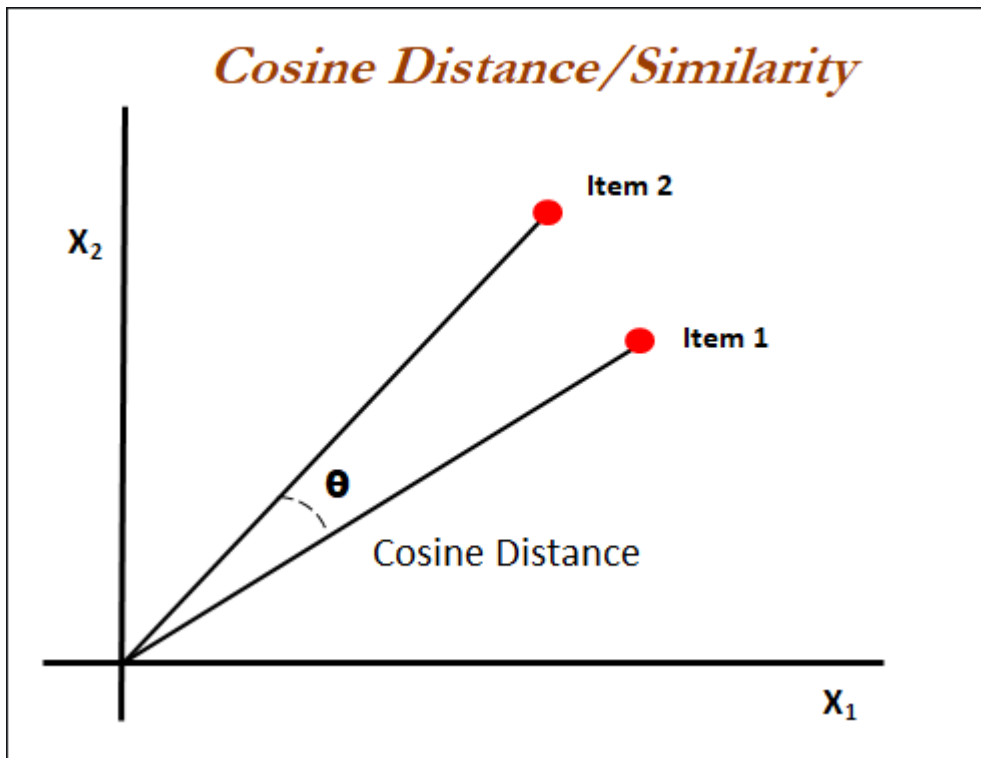
```
The Euclidean distance is:  1.4142135623730951
```

In [9]:
```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(I1, I2, 'ko--')
ax.annotate("Euclidean distance", xy=(1.8,2.8),  xytext=(0.5,0.3),textcoords='figure fraction',
            fontsize=16, arrowprops=dict(facecolor='black', shrink=0.1))
```

Out[9]: Text(0.5, 0.3, 'Euclidean distance')



We would actually need to calculate similarity using the angle between the line from the origin to Item1, and the line from the origin to Item2. To calculate similarity using an angle, we need a function that returns a higher similarity or smaller distance for a lower angle and a lower similarity or larger distance for a higher angle. The ***cosine*** of an angle is a function that decreases from 1 to -1 as the angle increases from 0 to 180, also known as ***cosine distance.*** In other words, -1 shows that two items are dissimilar and 1 shows that two items are completely similar.



You definitely remember the Unit Circle from high school.

The cosine similarity equation is:

$$\cos(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{AB}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

Where AB is the dot product of the two vectors and || A || || B || is the product of each vector's magnitude (or Eclidean distance from the origin to each vector).

Let us check the Cosine Similarity between Item1 and Item2. Observe below that the cosine similarity (or the angle == 0) is ~1 which means the items are very similar, as expected. Note that since all ranks are positive numbers (1 through 5), we will never have a negative cosine similarity value, bacause we will never have an angle greater that 90 degrees.

In [10]:
```
# Cosine Similarity calculated MANUALY
dot = np.array(I1).dot(np.array(I2))
E1 = spatial.distance.euclidean([0,0], I1)
E2 = spatial.distance.euclidean([0,0], I2)
sim_manual = dot / (E1*E2)
print ("Cosine Similarity MANUAL computation: ", sim_manual)
```

```
Cosine Similarity MANUAL computation:  0.9922778767136677
```

We can also use the cosine of the angle to find the similarity. The lower the angle, the higer will be the cosine and thus, the higher will be the similarity of the items. In our example the angle is almost 0, which means the items are very similar. We can also inverse the value of the cosine of the angle by subtracting it from 1 to get the cosine similarity from above between the items.

In [11]:
```
# An easier way to calculate
Cos = spatial.distance.cosine(I1,I2)
print ("The angle is: ", Cos)
sim = 1 - Cos
print ("The cosine similarity by subtracting the angle from 1 is: ", sim)
```

```
The angle is:  0.007722123286332261
The cosine similarity by subtracting the angle from 1 is:  0.9922778767136677
```

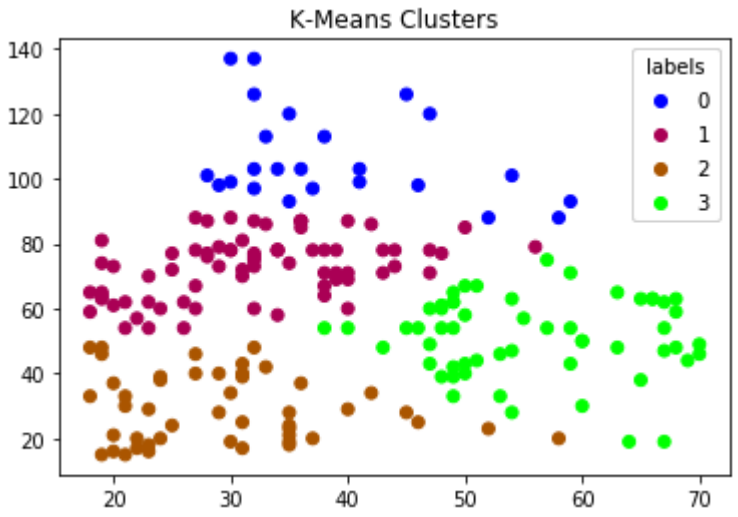
Now, let us try to build an item-to-item collaborative filtering recommendation system for one of the segments above.

Note that the model below is a somewhat simple approach. There are much more robust Deep Learning recommendation algorithms. Also, in addition to the Cold-Start probelm (as described above) for items that have not been ranked, we will generate random numbers as ratings. It means we will not capture the genuine and real sentiment expressed by real people. This will cause the model to be less accurate than it would be in reality, working with real customers giving real rankings. But the same method as below will be applied in reality.

Reperfom the K-Means algorithm

In [16]:
```
mall_customers = pd.read_csv(r"C:\Users\...\Downloads\Mall_Customers.csv")
mall_customers.rename(columns={'Annual Income (k$)': 'Annual Income'}, inplace=True)

df_final = mall_customers[['Age','Annual Income']]


X = df_final
kmeans = KMeans(n_clusters=4)
clusters = kmeans.fit_predict(X)

df_final['labels'] = kmeans.labels_
df_final['ID'] = mall_customers['CustomerID']
WCSS = kmeans.inertia_
centroids = kmeans.cluster_centers_

def plot(x, y, c):
    fig, ax = plt.subplots()
    scatter = ax.scatter(x, y, c=c, cmap=plt.cm.brg)
    legend = ax.legend(*scatter.legend_elements(),
                       loc="upper right", title="labels")
    ax.add_artist(legend)
    ax.set_title('K-Means Clusters')
    plt.show()

print ('WCSS: {}'.format(WCSS), '\nCentroids:\n {}'.format(centroids))
plot(df_final['Age'], df_final['Annual Income'], df_final['labels'])
```

```
WCSS: 39502.77850064538
Centroids:
 [[ 39.         106.5       ]
 [ 31.95890411  72.95890411]
 [ 29.44680851  29.21276596]
 [ 55.66071429  51.01785714]]
```



Select Segment #1

In [17]:
```
seg_1 = df_final[df_final['labels'] == 1]
rows = seg_1.shape[0]
print("There are", rows, "customers in segment 1")
cols = df_final.columns.tolist()
cols = cols[-1:] + cols[:-1]
seg_1 = seg_1[cols]
display(seg_1)
```

```
There are 73 customers in segment 1
```

	ID	Age	Annual Income	labels
75	76	26	54	1
78	79	23	54	1
84	85	21	54	1
87	88	22	57	1
88	89	34	58	1
...	...	...	...	...
171	172	28	87	1
172	173	36	87	1
173	174	36	87	1
175	176	30	88	1
177	178	27	88	1

73 rows × 4 columns

We will create 10 products (Product1 through Product10) and randomly generate ratings from 1 to 5 with 50% NaN values which basically represent either products that have not been purchased by a customer or have not been rated in the past. In essence, one of the things we attempt with item-to-item recommendation model is to predict the NaN values.

In [49]:
```
products = ['Product1', 'Product2', 'Product3', 'Product4', 'Product5', 'Product6', 'Product7', 'Product8',
            'Product9', 'Product10']
pred = pd.DataFrame(np.random.randint(1,6,size=(rows, 10)), columns=products)
for col in pred.columns:
    pred.loc[pred.sample(frac=0.5).index, col] = pd.np.nan
display(pred)
```

	Product1	Product2	Product3	Product4	Product5	Product6	Product7	Product8	Product9	Product10
0	NaN	NaN	3.0	2.0	NaN	4.0	NaN	NaN	NaN	NaN
1	5.0	NaN	4.0	1.0	NaN	NaN	4.0	NaN	2.0	NaN
2	2.0	5.0	5.0	NaN	NaN	3.0	1.0	5.0	NaN	NaN
3	NaN	2.0	2.0	NaN	1.0	5.0	NaN	5.0	4.0	5.0
4	NaN	5.0	NaN	NaN	5.0	1.0	5.0	5.0	4.0	NaN
...	...	...	...	...	...	...	...	...	...	...
68	NaN	NaN	1.0	3.0	NaN	1.0	NaN	NaN	NaN	1.0
69	NaN	NaN	NaN	NaN	1.0	4.0	NaN	1.0	1.0	1.0
70	5.0	2.0	NaN	NaN	NaN	3.0	1.0	3.0	1.0	2.0
71	4.0	1.0	4.0	1.0	2.0	NaN	NaN	4.0	1.0	2.0
72	NaN	4.0	1.0	NaN	2.0	3.0	1.0	NaN	3.0	NaN

73 rows × 10 columns

Join rating DataFrame with Segment #1 Customer ID

In [50]:
```
pred = pred.set_index(seg_1['ID'])
display(pred)
```

ID	Product1	Product2	Product3	Product4	Product5	Product6	Product7	Product8	Product9	Product10
76	NaN	NaN	3.0	2.0	NaN	4.0	NaN	NaN	NaN	NaN
79	5.0	NaN	4.0	1.0	NaN	NaN	4.0	NaN	2.0	NaN
85	2.0	5.0	5.0	NaN	NaN	3.0	1.0	5.0	NaN	NaN
88	NaN	2.0	2.0	NaN	1.0	5.0	NaN	5.0	4.0	5.0
89	NaN	5.0	NaN	NaN	5.0	1.0	5.0	5.0	4.0	NaN
...	...	...	...	...	...	...	...	...	...	...
172	NaN	NaN	1.0	3.0	NaN	1.0	NaN	NaN	NaN	1.0
173	NaN	NaN	NaN	NaN	1.0	4.0	NaN	1.0	1.0	1.0
174	5.0	2.0	NaN	NaN	NaN	3.0	1.0	3.0	1.0	2.0
176	4.0	1.0	4.0	1.0	2.0	NaN	NaN	4.0	1.0	2.0
178	NaN	4.0	1.0	NaN	2.0	3.0	1.0	NaN	3.0	NaN

73 rows × 10 columns

Next, convert all 'NaN' values to zero so we could work with the data and apply the cosine similarity.

In [51]:
```
pred = pred.fillna(0)
display(pred)
```

ID	Product1	Product2	Product3	Product4	Product5	Product6	Product7	Product8	Product9	Product10
76	0.0	0.0	3.0	2.0	0.0	4.0	0.0	0.0	0.0	0.0
79	5.0	0.0	4.0	1.0	0.0	0.0	4.0	0.0	2.0	0.0
85	2.0	5.0	5.0	0.0	0.0	3.0	1.0	5.0	0.0	0.0
88	0.0	2.0	2.0	0.0	1.0	5.0	0.0	5.0	4.0	5.0
89	0.0	5.0	0.0	0.0	5.0	1.0	5.0	5.0	4.0	0.0
...	...	...	...	...	...	...	...	...	...	...
172	0.0	0.0	1.0	3.0	0.0	1.0	0.0	0.0	0.0	1.0
173	0.0	0.0	0.0	0.0	1.0	4.0	0.0	1.0	1.0	1.0
174	5.0	2.0	0.0	0.0	0.0	3.0	1.0	3.0	1.0	2.0
176	4.0	1.0	4.0	1.0	2.0	0.0	0.0	4.0	1.0	2.0
178	0.0	4.0	1.0	0.0	2.0	3.0	1.0	0.0	3.0	0.0

73 rows × 10 columns

Since not all customers have the same standards and some are harder to please, those customers are most likely to be tough rating givers, which as are knowns as pessimistic reviewers. In other words, pessimistic reviewers are defined as biased reviewers who have given relatively low ratings to all products, including high-quality products and optimistic reviewers are those reviewers who have given relatively high ratings to all products that they have reviewed, including low-quality products.

We would have to address the pessimistic/ optimistic problem and standardize the ratings to account for these biases. To do that we would need to compute each rating's $Z - Score$ such that given the following ratings from User 1:

User 1: 2, 2, 5, 5

We compute the mean μ_i of User 1's rating and standar deviation σ_i, then subtract μ_i of row i from rating $x_{i,j}$ divided by σ_i to get:

$z_{i,j} = \frac{x_{i,j} - \mu_i}{\sigma i}$

Note that by doing the above each row's mean == 0 and standard deviation == 1.

User 1 (new rating calculated by z-score): -0.8660254 , -0.8660254 , 0.8660254 , 0.8660254

In [52]:
```
from scipy.stats import zscore

pred_scale = pd.DataFrame(zscore(pred, axis=1, ddof=1))
display(pred_scale)
```

	0	1	2	3	4	5	6	7	8	9
0	-0.590596	-0.590596	1.378058	0.721840	-0.590596	2.034276	-0.590596	-0.590596	-0.590596	-0.590596
1	1.690634	-0.795592	1.193388	-0.298347	-0.795592	-0.795592	1.193388	-0.795592	0.198898	-0.795592
2	-0.044771	1.298363	1.298363	-0.940194	-0.940194	0.402940	-0.492482	1.298363	-0.940194	-0.940194
3	-1.105731	-0.184289	-0.184289	-1.105731	-0.645010	1.197875	-1.105731	1.197875	0.737154	1.197875
4	-1.015928	1.015928	-1.015928	-1.015928	1.015928	-0.609557	1.015928	1.015928	0.609557	-1.015928
...	...	...	...	...	...	...	...	...	...	...
68	-0.621059	-0.621059	0.414039	2.484236	-0.621059	0.414039	-0.621059	-0.621059	-0.621059	0.414039
69	-0.650791	-0.650791	-0.650791	-0.650791	0.162698	2.603165	-0.650791	0.162698	0.162698	0.162698
70	2.016632	0.183330	-1.038871	-1.038871	-1.038871	0.794431	-0.427770	0.794431	-0.427770	0.183330
71	1.316506	-0.564217	1.316506	-0.564217	0.062691	-1.191124	-1.191124	1.316506	-0.564217	0.062691
72	-0.929896	1.726949	-0.265684	-0.929896	0.398527	1.062738	-0.265684	-0.929896	1.062738	-0.929896

73 rows × 10 columns

Let us now apply the cosine similarity between each product.

In [53]:
```
from sklearn.metrics.pairwise import cosine_similarity
pred_sim  = cosine_similarity(pred_scale.T)
cos_sim = pd.DataFrame(pred_sim, index=pred.columns, columns=pred.columns)
display(cos_sim)
```

	Product1	Product2	Product3	Product4	Product5	Product6	Product7	Product8	Product9	Product10
Product1	1.000000	-0.105400	-0.009537	0.026852	-0.114145	-0.288712	-0.060300	-0.081980	-0.249949	-0.161898
Product2	-0.105400	1.000000	0.033162	-0.341928	0.033654	-0.134491	-0.273420	0.057181	0.023592	-0.262126
Product3	-0.009537	0.033162	1.000000	-0.186379	-0.180935	-0.126256	-0.091818	-0.091379	-0.087547	-0.262358
Product4	0.026852	-0.341928	-0.186379	1.000000	-0.193379	0.008298	-0.117768	-0.194207	-0.189304	0.097923
Product5	-0.114145	0.033654	-0.180935	-0.193379	1.000000	-0.093356	-0.041929	-0.134102	0.007976	-0.279578
Product6	-0.288712	-0.134491	-0.126256	0.008298	-0.093356	1.000000	-0.184943	-0.198739	0.063214	-0.025070
Product7	-0.060300	-0.273420	-0.091818	-0.117768	-0.041929	-0.184943	1.000000	-0.241087	-0.207568	0.072660
Product8	-0.081980	0.057181	-0.091379	-0.194207	-0.134102	-0.198739	-0.241087	1.000000	0.033510	-0.021803
Product9	-0.249949	0.023592	-0.087547	-0.189304	0.007976	0.063214	-0.207568	0.033510	1.000000	-0.192241
Product10	-0.161898	-0.262126	-0.262358	0.097923	-0.279578	-0.025070	0.072660	-0.021803	-0.192241	1.000000

Now we are ready to recommend products to customers. Note that the 'Recommendation Score' below is merely an arbitrary score.

Let us try to recommend Customer ID 85 who gave Product2 five stars. Observe below that we first recommend Customer ID 85 Product8 which has the highest cosine similarity (0.057181) to Product2 then the other products. (Note that Customer ID 85 has already purchased Product8 and also gave it a 5 stars review.) We would want to recommend this product, because Customer ID 85 liked Product2, and Product8 is most similar to it. The next recommended product is Product5 which the customer has not purchased.

In [56]:
```
def recommend(Customer_ID, Product):
    rating = pred.loc[Customer_ID][Product]
    if rating == 0:
        print("Customer", Customer_ID, "has not purchased", Product)
    else:
        recommendation = cos_sim[Product]*(rating-2.5) # -2.5 for easier sorting in case we feed with 1 star
 review product
        recommendation = recommendation.sort_values(ascending=False)
        recommendation = recommendation.loc[recommendation.index != Product]
        recommendation = pd.DataFrame(recommendation)
        recommendation.columns = ['Recommendation Score']
        return display(recommendation)

product = 'Product2'
print(recommend(85, product))

print("Cosine Similarity: ")
display(cos_sim.loc[product].sort_values(ascending=False))
```

	Recommendation Score
Product8	0.142952
Product5	0.084135
Product3	0.082906
Product9	0.058980
Product1	-0.263501
Product6	-0.336227
Product10	-0.655314
Product7	-0.683550
Product4	-0.854820

```
Cosine Similarity:

Product2     1.000000
Product8     0.057181
Product5     0.033654
Product3     0.033162
Product9     0.023592
Product1    -0.105400
Product6    -0.134491
Product10   -0.262126
Product7    -0.273420
Product4    -0.341928
Name: Product2, dtype: float64
```

Let us try Customer ID 178. We know he/ she purchased Product3 and gave it one a star review. The model recommends Product10, which the customer has not purchased yet, and has the least similarity to Product3 (-0.262358). Remember that the customer gave Product3 one star so we would want to recommend a dissimilar product.

In [57]:
```
def recommend(Customer_ID, Product):
    rating = pred.loc[Customer_ID][Product]
    if rating == 0:
        print("Customer", Customer_ID, "has not purchased", Product)
    else:
        recommendation = cos_sim[Product]*(rating-2.5) # -2.5 for easier sorting in case we feed with 1 star
 review product
        recommendation = recommendation.sort_values(ascending=False)
        recommendation = recommendation.loc[recommendation.index != Product]
        recommendation = pd.DataFrame(recommendation)
        recommendation.columns = ['Recommendation Score']
        return display(recommendation)

product = 'Product3'
print(recommend(178, product))

print("Cosine Similarity: ")
display(cos_sim.loc[product].sort_values(ascending=False))
```

	Recommendation Score
Product10	0.393536
Product4	0.279568
Product5	0.271403
Product6	0.189384
Product7	0.137727
Product8	0.137069
Product9	0.131320
Product1	0.014306
Product2	-0.049744

```
Cosine Similarity:

Product3     1.000000
Product2     0.033162
Product1    -0.009537
Product9    -0.087547
Product8    -0.091379
Product7    -0.091818
Product6    -0.126256
Product5    -0.180935
Product4    -0.186379
Product10   -0.262358
Name: Product3, dtype: float64
```

SUMMARY

Boots, UK's leading pharmacy, health and beauty retailer, wanted to inspire its ~15 million loyalty card holders to feel more engaged with the brand and to increase revenue generated from those customers. Armed with Boots' loyal customers data, we were able to apply the PCA algorithm to reduce dimensionality and K-Means to perform customer segmentation so Boots' marketing department would be able to deploy targeted and tailored marketing campaigns, which by itself a strategy to increase sales. However, we took a more personalized approach and based on products reviews collected from the customers, we were able to build an Item-to-Item recommendation engine that proved to be effective and to contribute to the increase in revenue derived form Boots' loyal customers. **MISSION SUCCESS!!!**

- Note that Boots understands the complex model above will have to be maintained as new customers join the loyal customer program, more products are offered, and additional reviews are provided by the customers. Since Boots trusts our experience and expertise, they would like us to oversee this task which means recurring revenue for us. **DOUBLE MISSION SUCCESS!!!**