Introduction

`Unsupervised models` are trained models using data without labels. Instead of telling an unsupervised algorithm what it should be looking for in the data, the algorithm does the work itself, in a sense independently finding structure within the data.

`K-Means clustering` is an unsupervised learning algorithm that, as the name hints, finds a fixed number (k) of `clusters` in a dataset. A cluster is a group of data points that are grouped together due to similarities in their features. When using a `K-Means algorithm`, a cluster is defined by a `centroid`, which is a point (either imaginary or real) at the center of a cluster. Every point in a data set is part of the cluster whose centroid is most closely located. To put it simply, the algorithm finds k number of centroids, and then assigns all data points to the closest cluster, with the aim of keeping the centroids small.

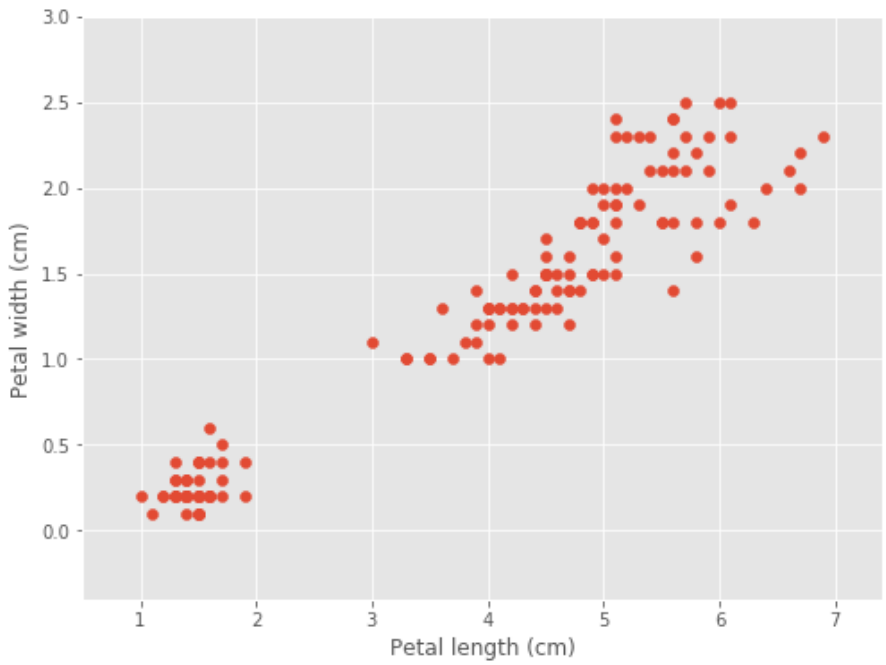
The Algorithm

We start off by randomly defining k `centroids` (can be an actual data point or it could be a random point). From there, the algorithm proceeds as follows:

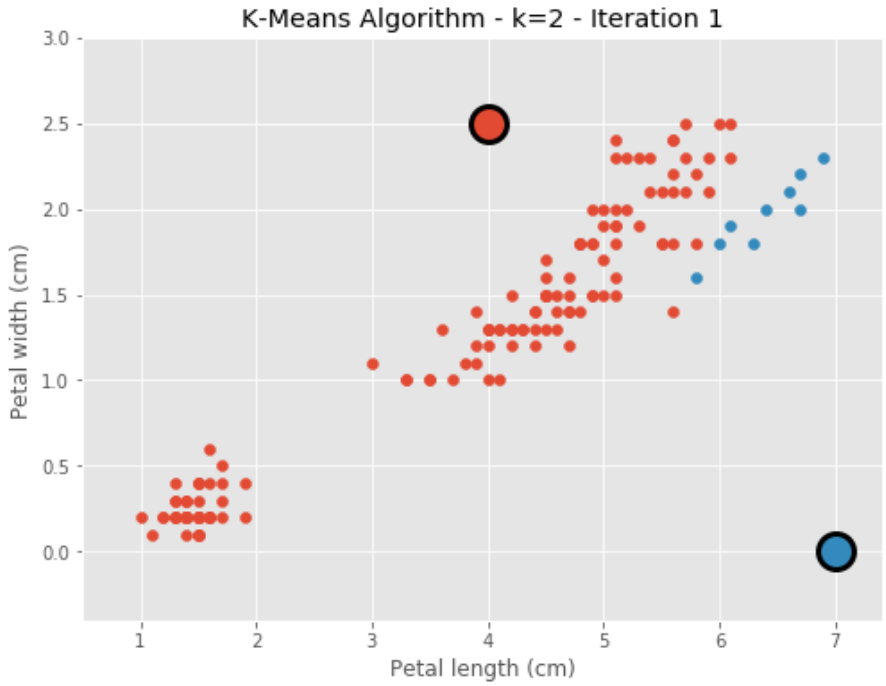
1. Assign each data point to the closest corresponding `centroid`, using the standard `Euclidean distance` (https://en.wikipedia.org/wiki/Euclidean_distance).
2. For each `centroid`, calculate the average of the values of all the points belonging to it. The mean value becomes the new value of the centroid.

Once step 2 is complete, all of the centroids have new values that correspond to the average of all of their corresponding points. This process is repeated over and over until there is no change in the centroid values, meaning that they have been accurately grouped. Or, the process can be stopped when a previously determined maximum number of steps has been met.

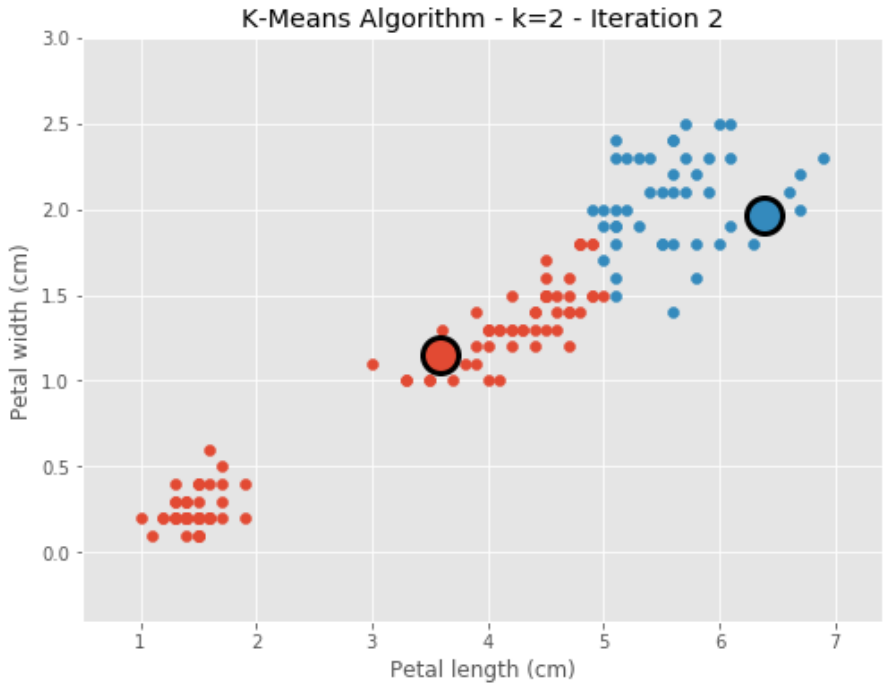
Let's assume we have a set of data points as below



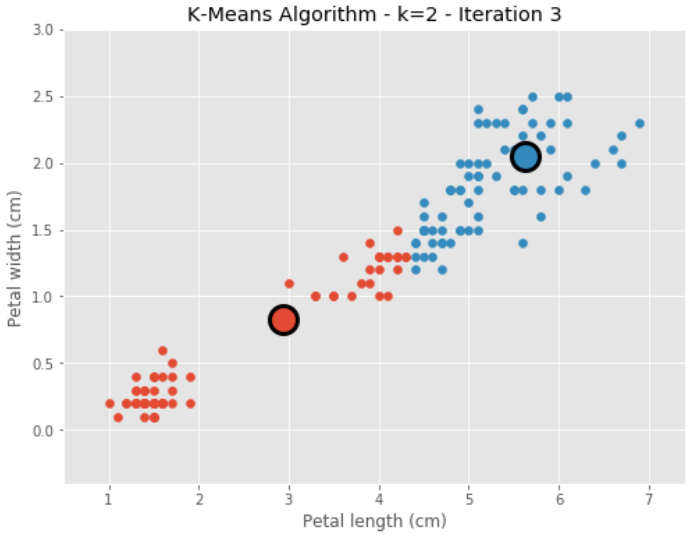
Iteration 1: First, we create two randomly generated centroids and assign each data point to the cluster of the closest centroid. In this case, because we are using two centroids, our k value is 2.

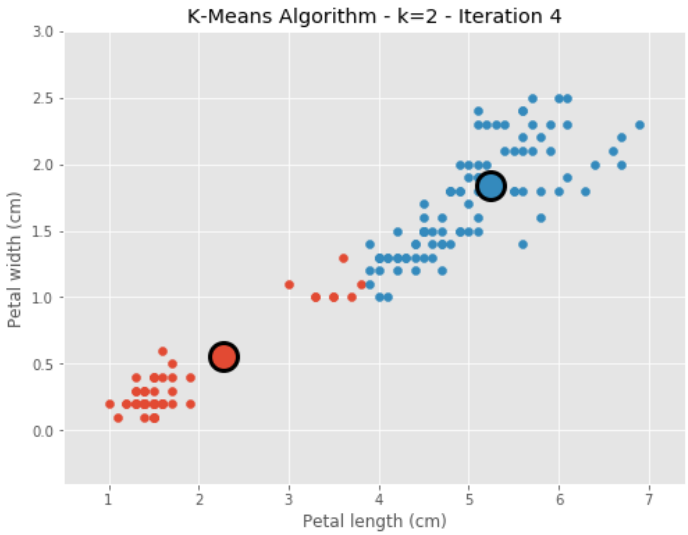


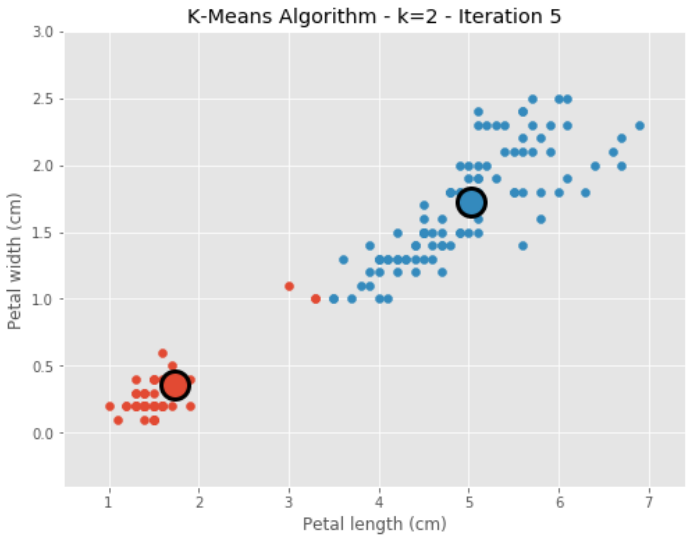
Iteration 2: As you can see above, the centroids are not evenly distributed. In the second iteration of the algorithm, the average values of each of the two clusters are found and become the new centroid values.



Iterations 3-5: We repeat the process until there is no further change in the value of the centroids.

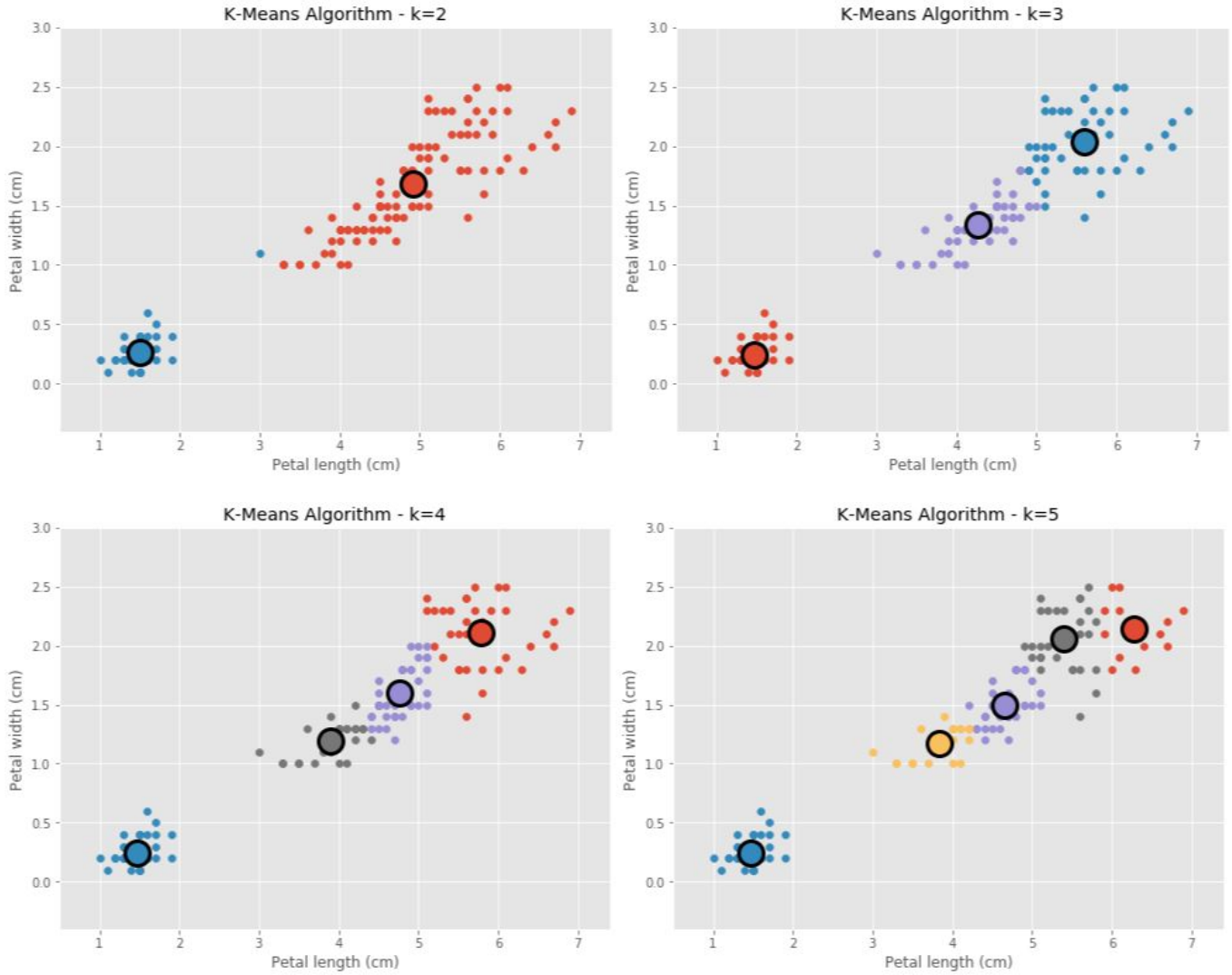






Finally, after iteration 5, there is no further change in the clusters.

Now, let's take a look at the visual differences between using $k = 2, 3, 4 \text{ or } 5$ clusters.



Choosing the most optimal K number of `clusters` depends on the problem at hand. There are many available to do so:

- The Elbow method (https://en.wikipedia.org/wiki/Elbow_method_(clustering)): This is probably the most well-known method for determining the optimal number of clusters. It is also a bit naive in its approach.
- The Silhouette Method (https://en.wikipedia.org/wiki/Silhouette_(clustering)): The silhouette value measures how similar a point is to its own cluster (cohesion) compared to other clusters (separation).

`The Elbow Method` is more of a decision rule, while the `Silhouette` is a metric used for validation while clustering. Thus, it can be used in combination with the Elbow Method. Both methods are not alternatives to each other for finding the optimal K, rather they are tools to be used together for a more confident decision.

Mathematical Explanation

Task: Implement clustering via $K-means$ and $K-medoids$ algorithms

$K-Means$ steps

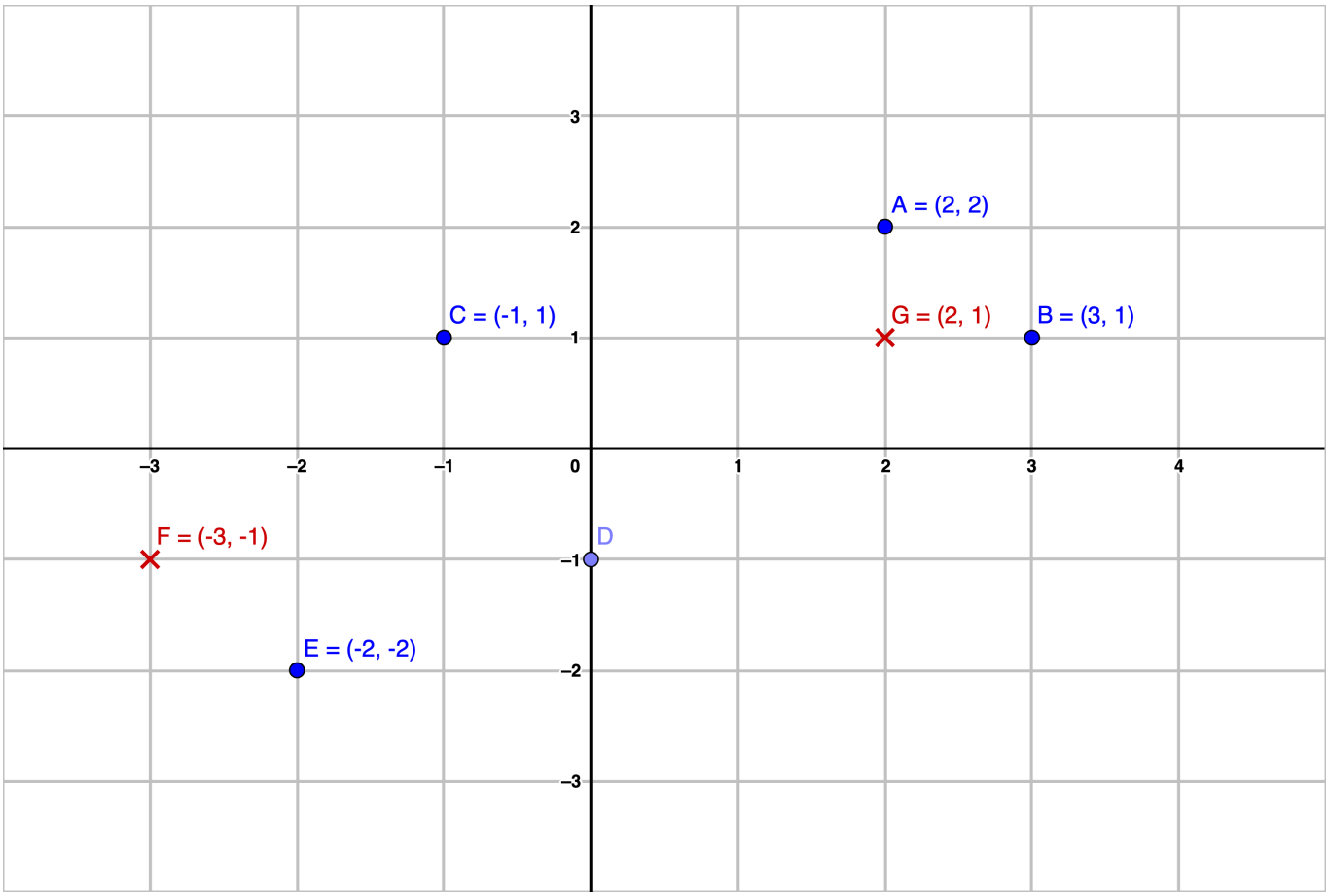
1. Given N data points, $\{x_1, x_2, \cdots, x_N\}$ arrays with real numbers
2. Find K cluster centers, $\{\mu_1, \mu_2, \ldots, \mu_K\}$
3. And assign each data point x_i to one cluster

$K-means$ algorithm can be implemented as follows:

- Initialize K cluster centers, $\{\mu_1, \mu_2, \ldots, \mu_K\}$, randomly
- Do
  - Decide the cluster memberships of each data point x_n by assigning it to the nearest cluster center (cluster assignment)
$$r_{nk} = \begin{cases} 1 & if\ k = argmin_j \|x_n - \mu_j\|^2 \\ 0 & otherwise. \end{cases}$$
  - Adjust each cluster center μ_k by setting it equal to the mean of all of the data points x_n assigned to cluster k (center adjustment)
$$\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$$
  - While any cluster center has been changed

Little exercise

Assume $K = 2$, say we have the following chart



1. Calculate the position of the two `centroids` after 1 iteration
2. Would the algorithm stop after the first iteration?

Coding Example: Image compression using K-means

In our problem of image compression, K-means algorithm will group similar colors together into K clusters of different colors (RGB values). Therefore, each `cluster centroid` becomes the representative of the three dimensional color vector in RGB color space of its respective cluster.

In [3]:

```
## loading relevant libraries
import sys
import time

import numpy as np
from numpy.random import seed
from numpy.random import randint
from numpy.random import choice

import pandas as pd
import seaborn as sns
sns.set();

from scipy.spatial.distance import cdist

import matplotlib.pyplot as plt
import matplotlib as mpl
from PIL import Image
from matplotlib.pyplot import imshow
plt.rcParams['figure.figsize'] = (16, 16)
plt.style.use('default')
%matplotlib inline

print('Python version:', sys.version)
print('Numpy version:', np.__version__)
print('Matplotlib version:', mpl.__version__)
```

```
Python version: 3.7.1 (default, Dec 14 2018, 13:28:58)
[Clang 4.0.1 (tags/RELEASE_401/final)]
Numpy version: 1.18.1
Matplotlib version: 3.1.0
```

In [4]:

```
## Initialising centroids
def init_centroid(X, k=5):
    seed(2020)
    samples = choice(len(X), size=k, replace=False)
    return X[samples, :]

## Distance calculations
def dist(X, C):
  return np.linalg.norm(X[:, np.newaxis, :] - C, ord=2, axis=2) ** 2

## Cluster assignment
def clst_assign(S):
  return np.argmin(S, axis=1)

## Update cluster
def update(X, clst):
  m, d = X.shape
  k = max(clst) + 1
  C_new = np.empty((k, d))
  for v in range(k):
    C_new[v, :d] = np.mean(X[clst == v, :], axis=0)
  return C_new

## Within cluster sum
def WithinClusterSSQ(S):
  return np.sum(np.amin(S, axis=1))

## Applying k-means
def kmeans(X, k=5, C_start=None, max_steps=np.inf, breakloop=False):
  if C_start is None:
    C = init_centroid(X, k)
  else:
    C = C_start
  converged = False
  clusters = np.zeros(len(X))
  labels = np.zeros(len(X))
  i = 1
  while (not converged) and (i <= max_steps):
    C_old = C
    S = dist(X, C)
    labels = clst_assign(S)
    C = update(X, labels)
    converged = np.array_equal(C_old, C)
    i += 1
    if i > 10 & breakloop == True:
      break
  return labels, i, WithinClusterSSQ(S)
```

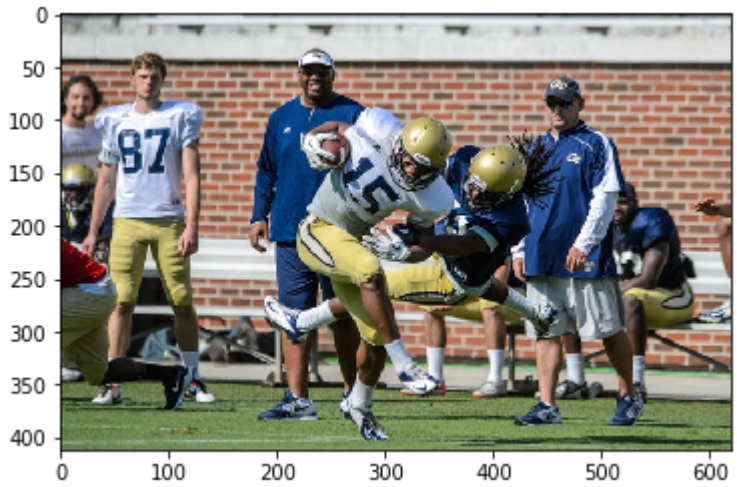
Loading image

In [5]:

```
img = Image.open('HW1_Data/football.bmp')
np_img = np.array(img, dtype='int32')
arr = np_img.astype(dtype='uint8')
img = Image.fromarray(arr, 'RGB')
imshow(np.asarray(img), aspect='auto')
```

Out[5]:

```
<matplotlib.image.AxesImage at 0x101e14278>
```



In [10]:

```
print("Dimensions of the matrix is: {} rows x {} columns x depth of {} ".format(
np_img.shape[0], np_img.shape[1], np_img.shape[2]))
```

```
Dimensions of the matrix is: 412 rows x 620 columns x depth of 3
```

One way we can view this set of pixels is as a cloud of points in a three-dimensional color space. We will reshape the data to `n_samples x n_features` , and rescale the colors so that they lie between 0 and 1:

In [9]:

```
## Reshape the array to (240 x 320) x 3
row, col, l = np_img.shape
data = np.reshape(np_img, (row*col, l), order='C')
m, k = data.shape
"The dimensions of our new data are: {} rows and {} columns" .format(m, k)
display(data[:10, :])
```

```
array([[76, 84, 87],
       [73, 81, 84],
       [68, 76, 79],
       [63, 71, 74],
       [58, 66, 69],
       [56, 64, 67],
       [55, 63, 66],
       [55, 63, 66],
       [52, 61, 60],
       [50, 59, 58]], dtype=int32)
```

Algorithm implementation

In [23]:

```
## Centroids initialization
C_inits = init_centroid(data)
print("Initial centroidss:\n", C_inits)
```

```
Initial centroidss:
 [[ 10  21  39]
 [225 227 224]
 [108 119  76]
 [121 146 200]
 [105 131  86]]
```

In [24]:

```
## Distance calculations
M = dist(data, C_inits)
print(M[100:104, ])
```

```
[[84770.  3614. 26083.  5961. 22574.]
 [85069.  3603. 26430.  5838. 22875.]
 [86409.  3361. 27230.  6062. 23617.]
 [87430.  3170. 27785.  6289. 24134.]]
```

In [25]:

```
## Cluster Assignment
cluster_labels = clst_assign(M)
print("cluster labels:", np.unique(cluster_labels))
```

```
cluster labels: [0 1 2 3 4]
```

In [26]:

```
## Update centroid values
centers = update(data, cluster_labels)
print(centers)
```

```
[[ 29.11412524  36.17941496  41.60845933]
 [211.23658052 210.03666888 204.44585479]
 [115.87709262  95.80146173  74.14971206]
 [164.59083378 159.23823895 159.16489141]
 [130.88841598 135.68419996 107.81678378]]
```

So far, we have been running the $K-$means algorithm of $K = 5$. What we may want to do is to find the optimal K value that maximises the *in-between* cluster distances and minimises the *within* cluster distances.

In [27]:

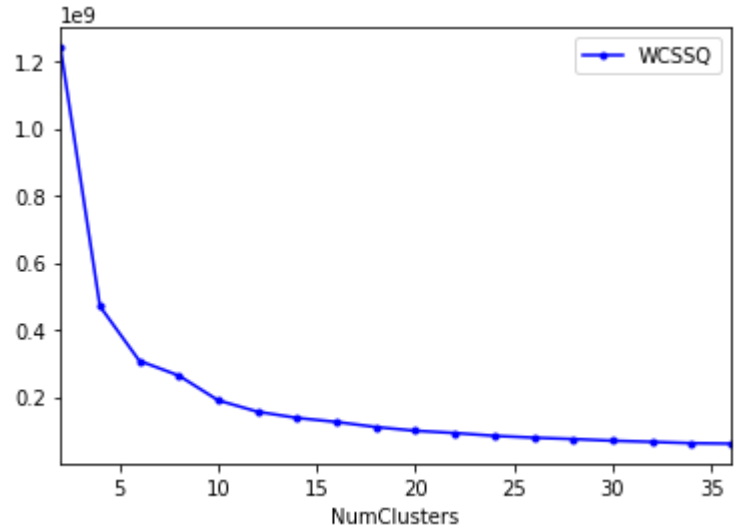
```
col_names = ['NumClusters', 'NumIterations', 'WCSSQ']
res_df = pd.DataFrame(columns = col_names)
for i in range(36):
    i += 1
    if i % 2 != 0:
        continue
    new_labels, iteration, WCSSQ = kmeans(data, k=i)
    res_df.loc[len(res_df)] = [i, iteration, WCSSQ]

res_df
```

Out[27]:

	NumClusters	NumIterations	WCSSQ
0	2.0	22.0	1.241175e+09
1	4.0	26.0	4.701919e+08
2	6.0	37.0	3.075240e+08
3	8.0	100.0	2.635508e+08
4	10.0	48.0	1.896540e+08
5	12.0	120.0	1.557243e+08
6	14.0	49.0	1.373021e+08
7	16.0	67.0	1.252524e+08
8	18.0	81.0	1.101027e+08
9	20.0	74.0	9.898811e+07
10	22.0	296.0	9.231399e+07
11	24.0	296.0	8.393195e+07
12	26.0	184.0	7.841276e+07
13	28.0	118.0	7.434619e+07
14	30.0	127.0	6.931669e+07
15	32.0	91.0	6.586664e+07
16	34.0	301.0	6.201605e+07
17	36.0	261.0	6.062951e+07

In [28]:

```
ax = plt.gca()
res_df.plot(style='.-',x='NumClusters',y='WCSSQ', color='blue', ax=ax)
plt.show()
```



Using the `Elbow-method` , we can confidently establish that the optimal $K = 5$ for our $K-$means algorithm. Let's have a look at our compressed image:

In [13]:

```
new_labels, iteration, WCSSQ = kmeans(data, k=5)
ind = np.column_stack((data, new_labels))
centers = {}
for i in set(new_labels):
    c = ind[ind[:,3] == i].mean(axis=0)
    centers[i] = c[:3]

img_clustered = np.array([centers[i] for i in new_labels])
img_disp = np.reshape(img_clustered, (row, col, l), order="C")
img_disp = img_disp.astype(dtype='uint8')
new_img = Image.fromarray(img_disp, 'RGB')

fig = plt.figure(figsize=(16, 16))
ax1 = fig.add_subplot(1,2, 1)
imshow(np.asarray(img))
ax1.title.set_text('Original Image')
ax2 = fig.add_subplot(1,2, 2)
imshow(np.asarray(new_img))
ax2.title.set_text('Compressed Image')
plt.show(block=True)
```