Clustering

Clustering is a type of unsupervised learning in which we try to find the features that are similar and group them. This group is referred to as a cluster. We mostly use L2 distance as a metric to cluster the features

Kmeans and Kmeans++

Kmeans

Kmeans is one of the simplest and most popular clustering algorithim that uses distance between feature points to create clusters. Normal kmeans clustering randomly assigns K points in feature space as cluster centers. Then i_th cluster is assigned a point if, the distance between the point and cluster i is minimum as compared to all other cluster centers.

Once all points are clustered, mean of each cluster is calculated which becomes new cluster centers. This process is repeated till new cluster and cluster in previous iteration are almost similar to each other or certain number of iterations has been completed.

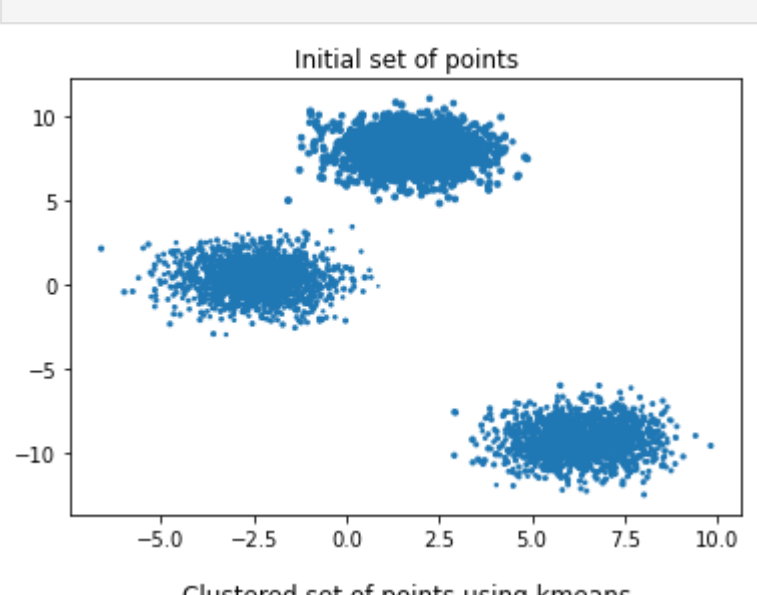
Kmeans++

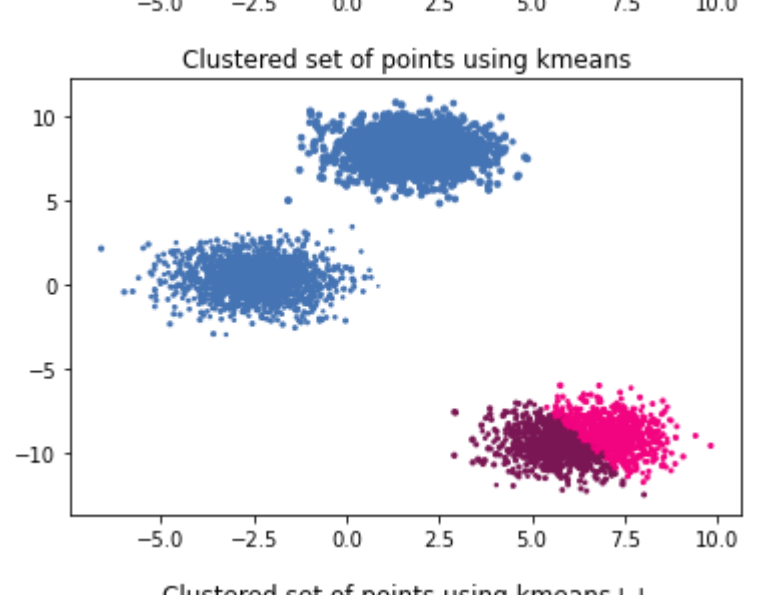
Kmeans++ is similar to Kmeans with the only difference in the procedure to intialize starting cluster centers. In Kmeans++ instead of randomly initializing K cluster centers, we randomly initalize 1 cluster center. To initialize rest of the cluster centers following steps are performed

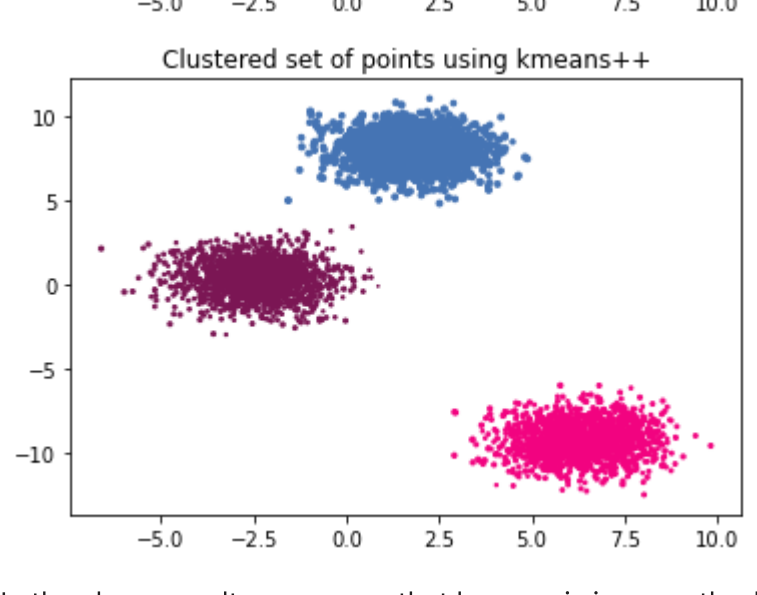
- First choose minimum distance of all the points with all the cluster centers.
- From the choosen minimum distances we choose the point with the maximum distance as the another cluster center.
- Repeat above two steps till we have acquired K cluster centers.

Kmeans++ ensures more intelligent initialization of cluster centers hence leads to better results as compared to kmeans. Below experiment justifies this statement

In [5]:
```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from kmeans import *
X, y = make_blobs(n_samples=5000, centers=3, n_features=3,random_state=20)
plt.scatter(X[:,0],X[:,1],X[:,2])
plt.title('Initial set of points')
plt.show()

centroids, labels = kmeans(X, 3)
plt.title('Clustered set of points using kmeans')
colors=np.array(['#4574B4','#F20380','#7B1654'])
plt.scatter(X[:,0], X[:,1],X[:,2], c=colors[labels])
plt.show()

centroids, labels = kmeans(X, 3,centroids='kmeans++')
plt.title('Clustered set of points using kmeans++')
colors=np.array(['#4574B4','#F20380','#7B1654'])
plt.scatter(X[:,0], X[:,1],X[:,2], c=colors[labels])
plt.show()
```







In the above result we can see that kmeans is incorrectly clustering the data points, whereas clusters created by kmeans++ looks correct.

Applications-

Kmeans++ has various applications like identifying crime localities, insurance fraud detection, customer segmentation etc. In this document we will look into following three application:

```
a. Breast Cancer Detection
b. Image compression
c. Facial feature detection
```

Breast cancer detection

In below experiment we use breast cancer dataset present in sklearn to categorize feature into benign and malignant cancer type

In [6]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
import numpy as np
import statistics



cancer = load_breast_cancer()
X = cancer.data
y = cancer.target

sc = StandardScaler()
X = sc.fit_transform(X)

centroids, labels = kmeans(X,k=2,centroids='kmeans++',tolerance=0.1)

labels=np.array(labels)
y = np.array(y)

ind_0 = labels==0
ind_1 = labels==1
```

Since cluster labels are arbitary created, we need to associate a cluster label to one of the categories. In below steps we try to identify mode of labels present in y corresponding to the indices of same cluster in labels

In [7]:
```
statistics.mode(y[ind_0])
```

Out[7]: 1

In [8]:
```
statistics.mode(y[ind_1])
```

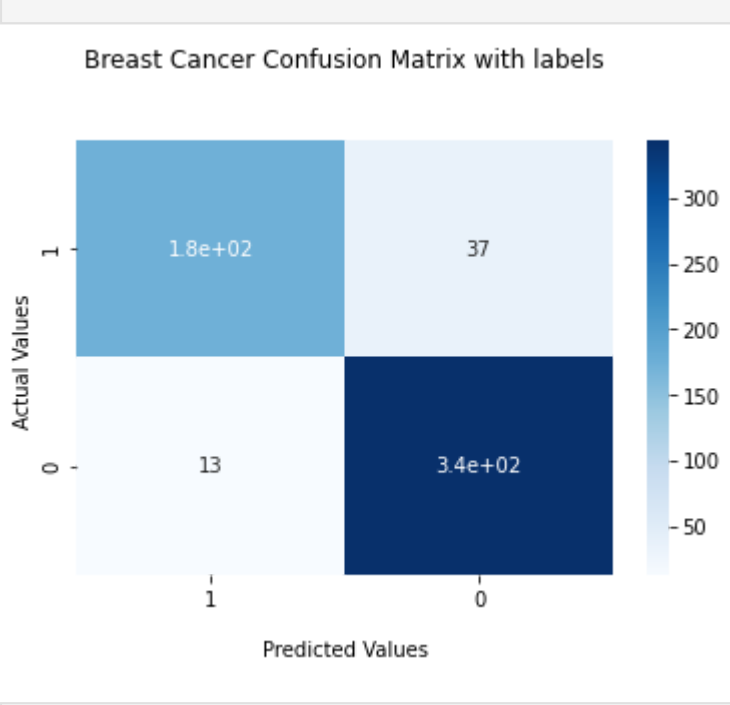
Out[8]: 0

Since cluster with label 1 represents y=0 and cluster with label 0 represents y=1. We change 1s in labels to 0s and 0s in labels to 1s so that cluster label becomes coherent with actual label

In [9]:
```
labels[labels==0]=-1
labels[labels==1]=0
labels[labels==-1]=1
```

Plotting confusion matrix

In [10]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

cf_matrix = confusion_matrix(y, labels)



ax = sns.heatmap(cf_matrix, annot=True, cmap='Blues')

ax.set_title('Breast Cancer Confusion Matrix with labels\n\n');
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values ');

## Ticket labels - List must be in alphabetical order
ax.xaxis.set_ticklabels(['1','0'])
ax.yaxis.set_ticklabels(['1','0'])

## Display the visualization of the Confusion Matrix.
plt.show()
```



In [11]:
```
Accuracy = (np.sum(np.eye(2)*cf_matrix)/np.sum(cf_matrix))*100
Accuracy
```

Out[11]: 91.21265377855887

In above result we have succesfully achived accuracy of 91.21%

Image Compression

Kmean++ clustering is a good way to compress an image without much compromising on the quality of image. In the below example we are trying to cluster an image on just 4 clusters and compare the size of initial image and image after clustering

Since we are using only 4 clusters final image might not be as sharp as initial image, but on increasing number of clusters, we can achieve sharper image at the cost of image size

In [12]:
```
import cv2
from PIL import Image
from skimage import io
img_ = Image.open('north-africa-1940s-grey.png')
io.imsave('north_africa_uncompressed_image.png', np.array(img_))
# image = cv2.imread('north-africa-1940s-grey.png')
```

Original Image

In [13]:
```
img_
```

Out[13]:

Clustering Begins

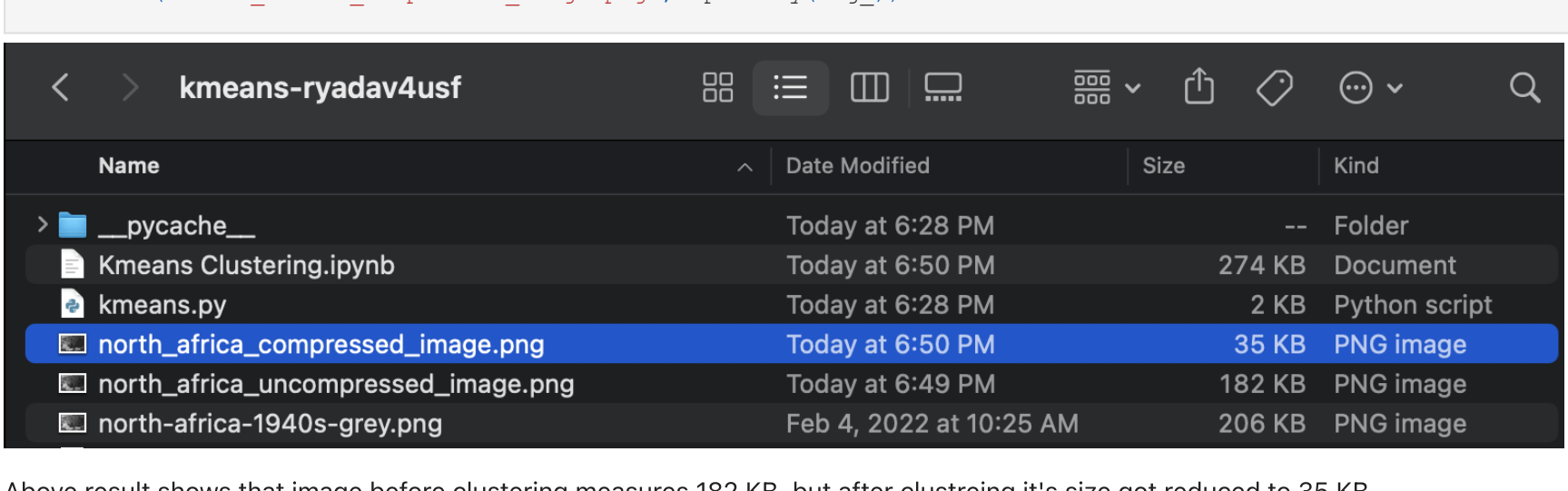
In [14]:
```
from kmeans import *
k=4
img_ = np.array(img_)
# X = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
h,w=img_.shape
centroids, labels = kmeans(img_.reshape(-1,1), k=k,centroids='kmeans++', tolerance=0.01)
centroids = centroids.astype(np.uint8)
img_ = centroids[labels] # reassign all points

img_ = Image.fromarray(img_.reshape(h,w), 'L')
img_
```

Out[14]:

Here we can see that even with 4 clusters final image looks quite similar to the original image

Saving compressed image

In [15]:
```
io.imsave('north_africa_compressed_image.png', np.array(img_))
```



Above result shows that image before clustering measures 182 KB, but after clustreing it's size got reduced to 35 KB

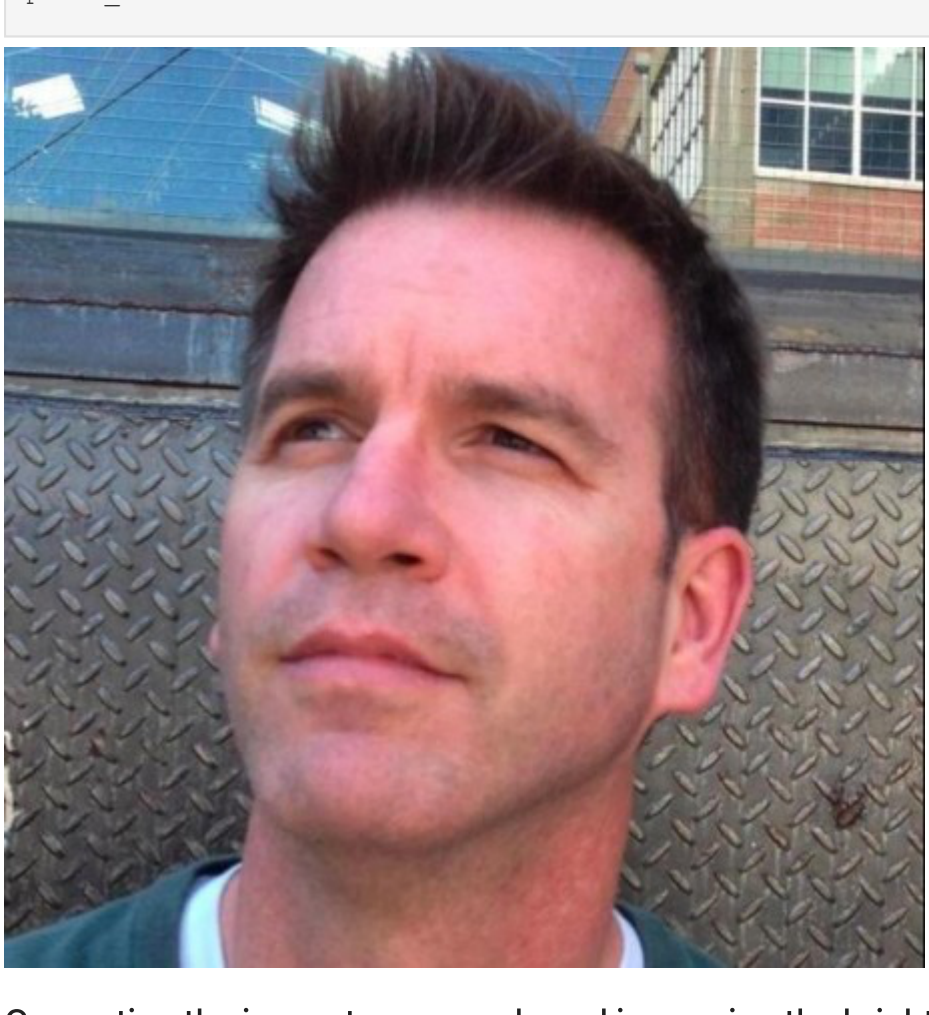
Facial feature detection

Facial feature detection requires a lot of image preprocessing so that edge detection algorithim can figure out correct edges. Kmeans++ helps in this process as it figures out the various regions in the face for example eyes,nose,jawline etc. If we use K=2, it pops up relevant regions, which makes the job of edge detection algorithim easier

Below is the image of Professor Terrance Parr. We will try to figure out the facial features in this image

In [16]:
```
import cv2
from PIL import Image
import numpy as np
parrt_ = Image.open('parrt_young.jpeg')
parrt_
```

Out[16]:



Converting the image to gray scale and increasing the brightness

In [17]:
```
import numpy as np
from PIL import Image,ImageFilter

def inc_brightness(img,brightness):
    brightness = brightness

    img = np.int16(img)
    img = img + brightness
    img = np.clip(img, 0, 255)
    img = np.uint8(img)
    return img

proc_parrt = cv2.cvtColor(inc_brightness(np.array(parrt_),100), cv2.COLOR_BGR2GRAY)

parrt_bw_ = Image.fromarray(proc_parrt, 'L')
parrt_bw_
```

Out[17]:

Trying to find edges in above image

In [18]:
```
parrt_bw_.filter(ImageFilter.FIND_EDGES)
```

Out[18]:

In above step we can see that on applying edge detection on raw black and white image, edge detection algorithim is not able to figure out facial features well

Now we are applying kmeans clustering with k=2 on the black and white image

Clustering begins

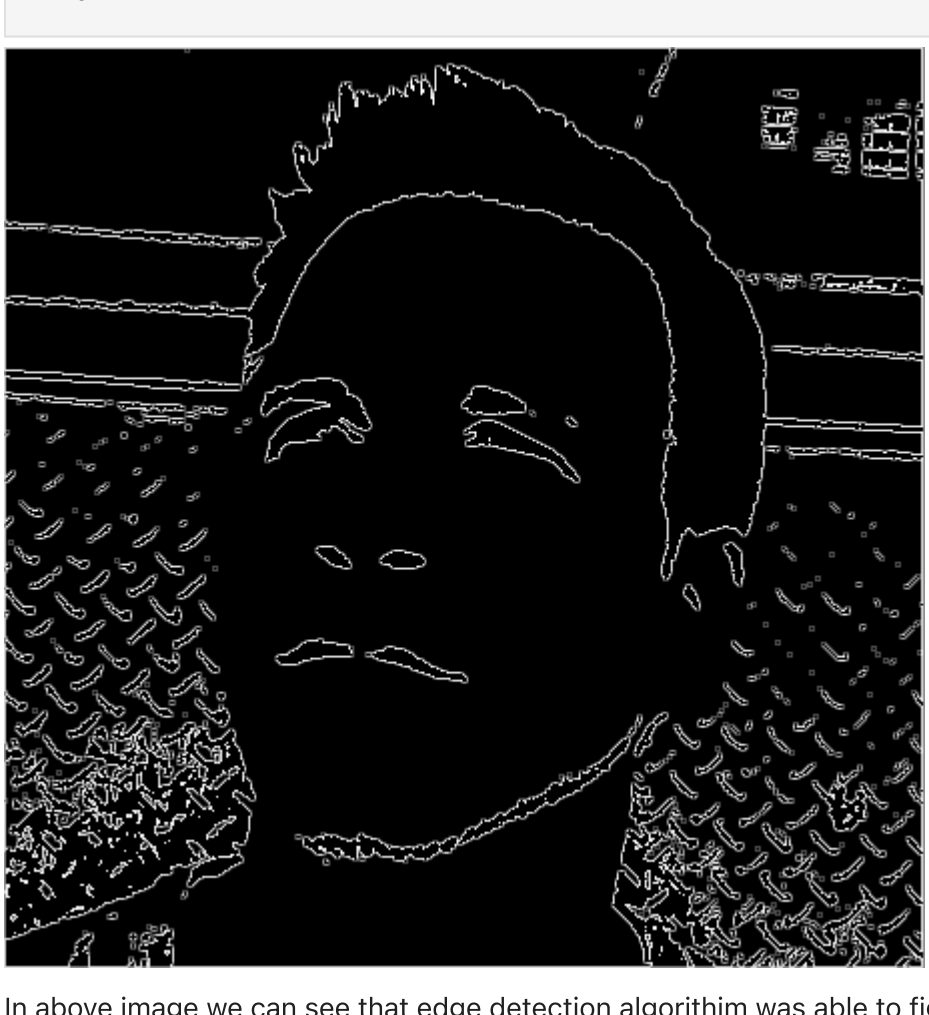
In [19]:
```
from kmeans import *
k=2
proc_parrt = cv2.cvtColor(inc_brightness(np.array(parrt_),100), cv2.COLOR_BGR2GRAY)
parrt_bw_ =proc_parrt
for _ in range(1):

    h,w=parrt_bw_.shape
    centroids, labels = kmeans(parrt_bw_.reshape(-1,1), k=k,centroids='kmeans++', tolerance=1)
    centroids = centroids.astype(np.uint8)
    parrt_bw_ = centroids[labels].reshape(h,w) # reassign all points

img_ = Image.fromarray(parrt_bw_, 'L')
img_
```

Out[19]:

Trying to find edges in above image

In [20]:
```
filtr=img_.filter(ImageFilter.FIND_EDGES)
filtr
```

Out[20]:

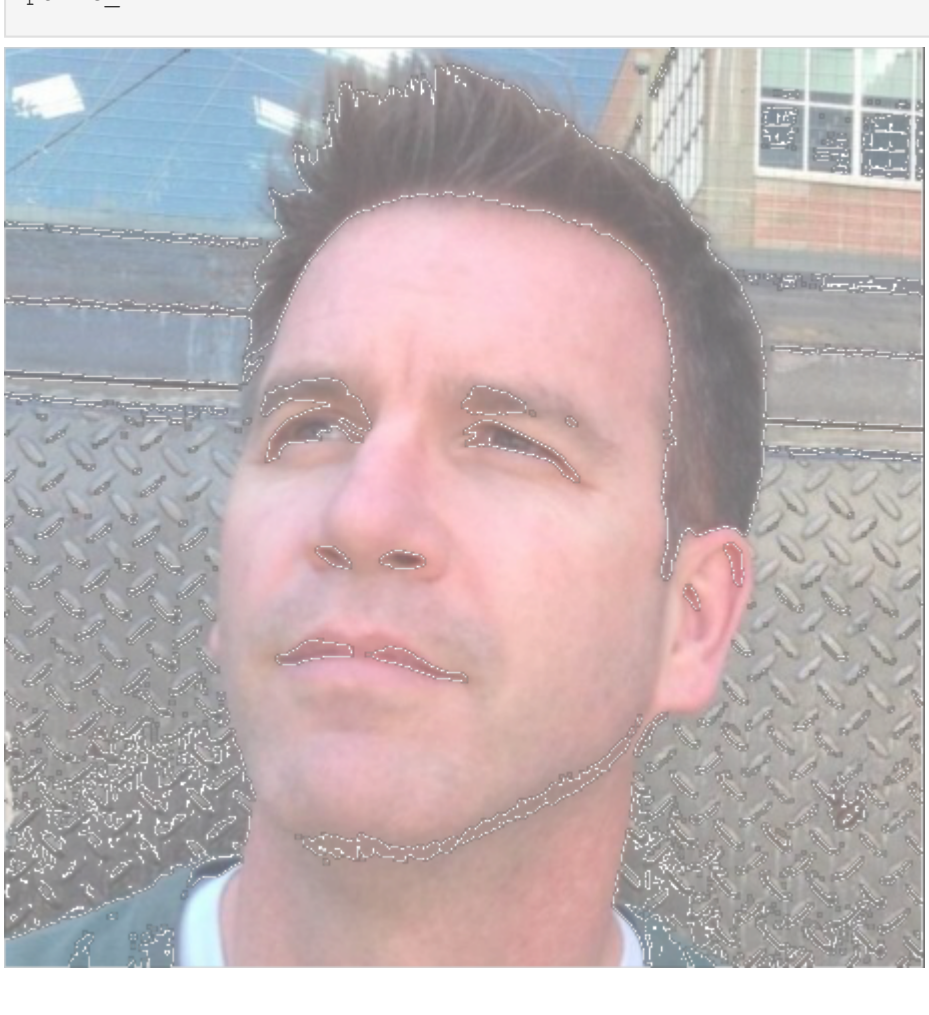


In above image we can see that edge detection algorithim was able to figure out facial features to a great extent. This suggests kmeans algorithim can be used to find facial features.

Superimposing the edges on real image

In [21]:
```
parrt_.putalpha(150)
parrt_.paste(filtr,(0, 0), filtr)
parrt_
```

Out[21]:



Limitation of Kmeans:

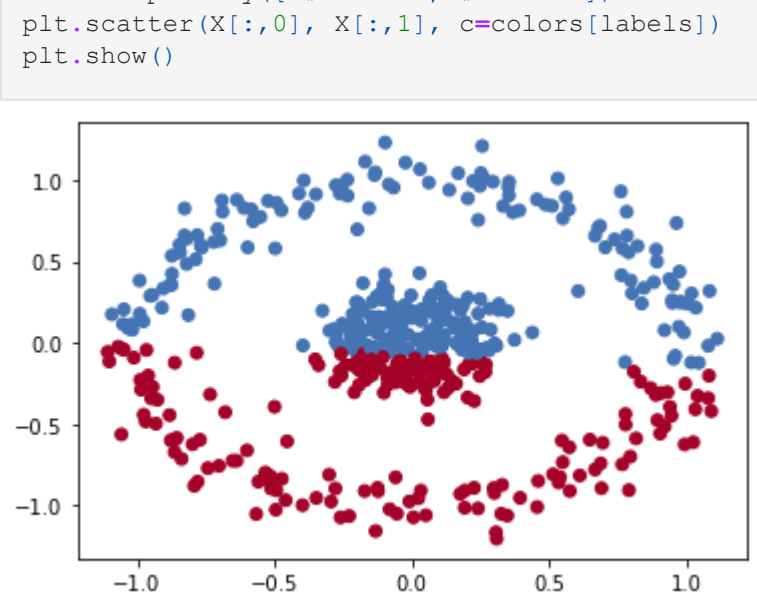
Even though Kmeans has lot of applications in practical life, it also has few limitations

a. Not good with huge data - Kmeans requires distances to be calculated from cluster center to each and every point. This process becomes computationally expensive when we are dealing with huge amount of data. Hence Kmeans is preferred only with small amount of data

b. Works with numerica data - Since Kmeans uses distance between points in feature space. Since it is tough to calculate distance between categorical features, we generally dont get good result with categorical data

c. Well suited for data with spherical clusters - Kmeans fails to recognise the clusters effectively if the clusters deviates from spherical shape. It's example is demonstrated below. In this example we try to compare clustering result in an image with non spherical clusters vs an image with spherical clusters

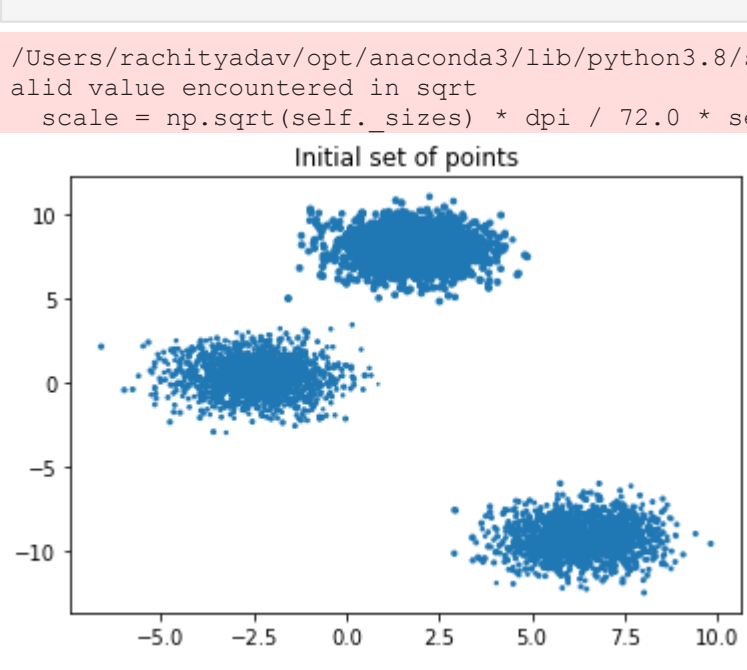
In [22]:
```
from sklearn.datasets import make_circles

from kmeans import kmeans


X, _ = make_circles(n_samples=500, noise=0.1, factor=.2)
centroids, labels = kmeans(X, 2, centroids='kmeans++')
colors=np.array(['#4574B4','#A40227'])
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()
```

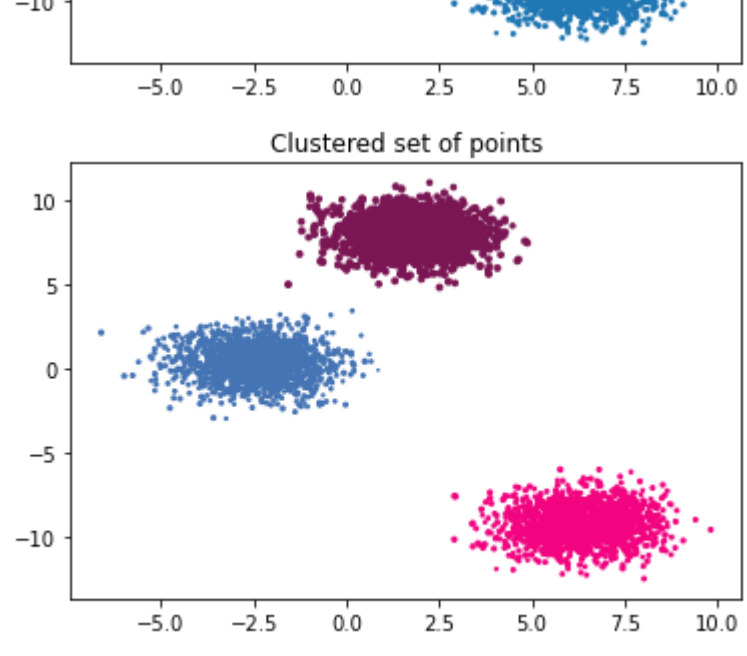


In the above example, we can see that kmeans is not able to identify inner circle and outer ring. Instead, it clusters half of the entire image as one cluster and another half as another cluster

In [23]:
```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=5000, centers=3, n_features=3,random_state=20)
plt.scatter(X[:,0],X[:,1],X[:,2])
plt.title('Initial set of points')
plt.show()
centroids, labels = kmeans(X, 3, centroids='kmeans++')
plt.title('Clustered set of points')
colors=np.array(['#4574B4','#F20380','#7B1654'])
plt.scatter(X[:,0], X[:,1],X[:,2], c=colors[labels])
plt.show()
```

```
/Users/rachityadav/opt/anaconda3/lib/python3.8/site-packages/matplotlib/collections.py:922: RuntimeWarning: inv
alid value encountered in sqrt
  scale = np.sqrt(self._sizes) * dpi / 72.0 * self._factor
```





In above example we can see the Kmeans has correctly clustered the data points.

This suggests performance of Kmeans is best when the shape of cluster is kind of spherical

Conclusion

Kmeans is one of the most popular clustering algorithim and is usually the first choice while implementing clustering. It initally chooses random centroids then iteratively shifts them and eventually, arrives at a set of centroids that are at almost the same location as the centroid in previous iteration. All features that are closest to a specific centroid forms a cluster.

Kmeans works well only when we have kind of spherical clusters in the data. If the cluster present is not spherical then kmeans does not gives correct result. Also, kmeans requires number of clusters to be pre defined. Therefore, we can say that kmeans clustering doesnt work in all scenarios, but in the scenarios where it does work, kmeans gives really good results.

In [ ]: