Kmeans Clustering

Author: Arman Tavana

In this notebook we talk about how k means clustering was implemented using (please find the implementation in the repo) and also show some examples of k means clustering using our implementation.

In [6]:
```
import pandas as pd
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler

from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.cluster import SpectralClustering
from kmeans import *
```

In [7]:
```
# %run kmeans
```

Intro

- Kmeans is a machine learning algorithm that does unsupervised learning. Unsupervised learning is that the algorithm makes clusters/groups using the vectors and not the labels or names in the dataset.
- One of the main hyperparameters in Kmeans algorithm is k. k is the number of clusters (centroids) you are looking for in your dataset.

Here is an example of using kmeans clustering with k=2 to find 2 clusters of points in the dataset.

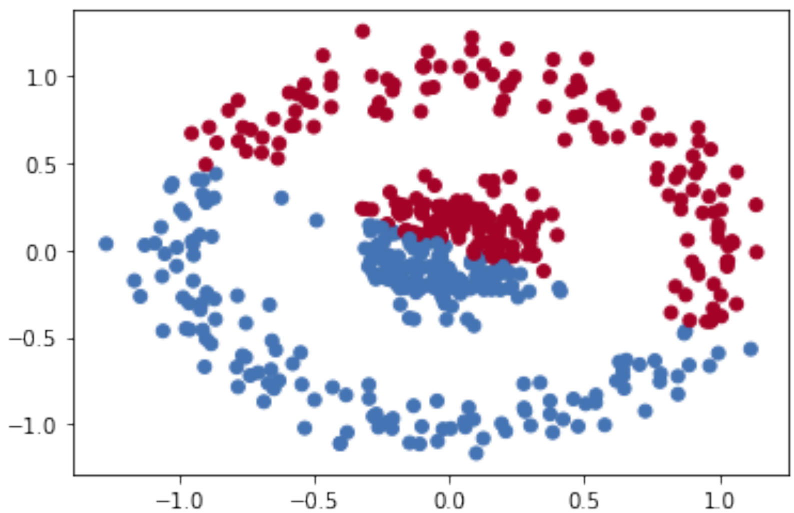
Implementation

1. First, we start by initializing our centroid. As mentioned above we use k++ technique to do that. In k++ we initilize the first centroid randomly and after that we pick the points (k-1 points) that maximize the minimum distance to all the other points to the clusters that we have in our data.
2. After that for each point we try to find the closest cluster and once we find the clusters we add the point to the nearest cluster.
3. In the next step we recompute our centroids again.
4. We repeat step 2 and 3 until our centroids stop changing or until we do maximum number of iteration (hyperparameter).

Example 1

In [8]:
```
X, _ = make_circles(n_samples=500, noise=0.1, factor=.2)
centroids, labels = kmeans(X, 2, centroids='kmeans++')
print(centroids)
colors=np.array(['#4574B4','#A40227'])
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()
plt.savefig("nested-kmeans.png", dpi=200)
```

```
[[-0.16648962 -0.34028139]
 [ 0.18164161  0.37105774]]
```




```
<Figure size 432x288 with 0 Axes>
```

- As we can observe in the example above there is a distinct line between the blue and red points. That has happened due to the centroids that our model has found using k++ initialization method for centroids.
- As we can see we have 2 centroids since we defined our k to be equal to 2.

Example 2

In [10]:
```
grades = [92.65, 93.87, 74.06, 86.95, 92.26, 94.46, 92.94, 80.65, 92.8
          93.03]
k = 3
grades = np.array(grades).reshape(-1, 1)
centroids, labels = kmeans(grades, k, centroids="kmeans++")
print("centroids", centroids.reshape(1, -1))
print("labels", labels)
```

```
centroids [[93.25222222 74.06       85.745     ]]
labels [0, 0, 1, 2, 0, 0, 0, 2, 0, 2, 0, 0, 2, 2, 2, 0]
```

In the example above I am trying to show what k means clustering looks like for numeric output. We have set bunch of arbitrary grades from around 70 to around 90. We defined our k to be 3 which means having three clusters.

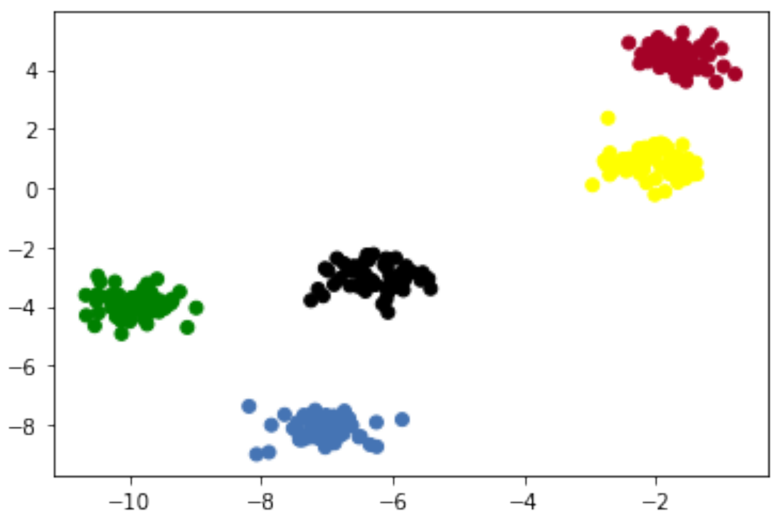
As expected we have 3 centroids that are around mid 70, 80, and 90. Also by looking at the labels we can see which clusters(labels) each data points belong to.

Example 3

In the example below there is a a good visualization to show how k means is able to detect different clusters in the data point and label them differently.

In [19]:
```
X, _ = make_blobs(n_samples=300, centers=5, cluster_std=0.40, random_s
centroids, labels = kmeans(X, 5, centroids='kmeans++')
print(centroids)
colors=np.array(['#4574B4','#A40227', 'green', 'yellow', 'black'])
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()
plt.savefig("nested-kmeans.png", dpi=200)
```

```
[[-7.06245396 -8.14417712]
 [-1.6291819   4.43042653]
 [-9.93145017 -3.93775512]
 [-2.07873105  0.83626738]
 [-6.28423845 -3.02433558]]
```



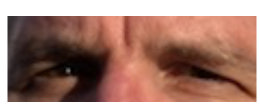

```
<Figure size 432x288 with 0 Axes>
```

Image Compression

Image 1

In here I am going to use k means clustering to perfome image compression. We can use k means clustering to compress images by using less colors than the original image and reproduce it with less colors.

Original Image



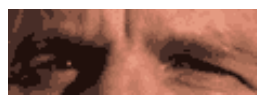
In [16]:
```
image = Image.open('eyes.png')
h = np.asarray(image).shape[0]
w = np.asarray(image).shape[1]
X = np.asarray(image).flatten().reshape(h*w,3)
```

In [17]:
```
k=10
centroids, labels = kmeans(X, k=k, centroids='kmeans++', tolerance=.01
centroids = centroids.astype(np.uint8)
X = centroids[labels]
```

In [18]:
```
img_ = Image.fromarray(X.reshape(h, w, 3))
img_.show()
```

In [63]:
```
img_.save("beautiful_eyes.png")
```

Compressed Version



In this example our algorithm was able to reproduce a compressed image with less colors (k=4). As we can see the model was able to do a decent job and reproduce these beautiful eyes.

How is K means clustering compressing images?

- An image is made of pixels and each pixel (colored) is 3 bytes of RGB (red, green, blue). RGB values are between 0 and 255 (each of them). Due to this attribute K means clustering is able to cluster similar colors together and create a compressed version of that image with less color. So that is how we were able to recreate an image with only 4 colors. We basically created 4 centroids and all the colors close to each centroid became a part of that centroid (the centroid near them).

Image 2

Since Roger Federe was my idol growing up and while I was playing professional tennis. I decided to recreate an image of him lifting his 20th grandslam (Australian Open 2018). #goat

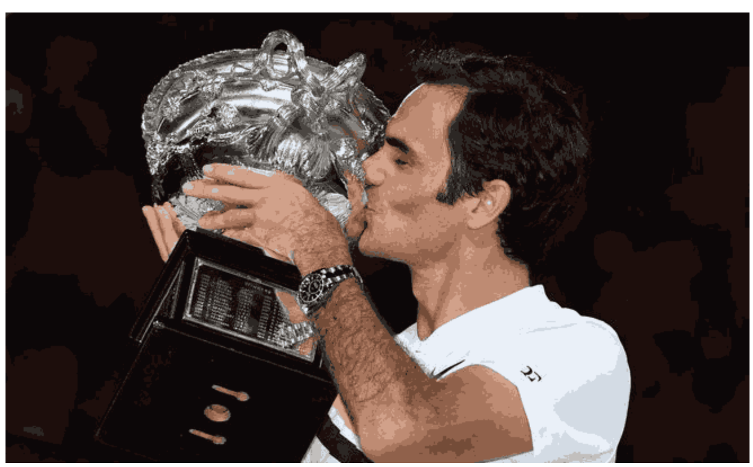
Original Image

In [68]:
```
image = Image.open('goat.jpg')
h = np.asarray(image).shape[0]
w = np.asarray(image).shape[1]
X = np.asarray(image).flatten().reshape(h*w,3)
```

In [69]:
```
k=15
centroids, labels = kmeans(X, k=k, centroids='kmeans++', tolerance=.01
centroids = centroids.astype(np.uint8)
X = centroids[labels]
```

In [70]:
```
img_ = Image.fromarray(X.reshape((h, w, 3)))
img_.show()
```

In [72]:
```
img_.save("compressed_goat.png")
```



Flaws of K means clustering

In the first example, kmeans clustering does not seem to be accurate since the data is a circle data and probabaly it's best to cluster each circle together. Since Kmeans clustering failed to do a decent job with the first example we are going to explore other options.

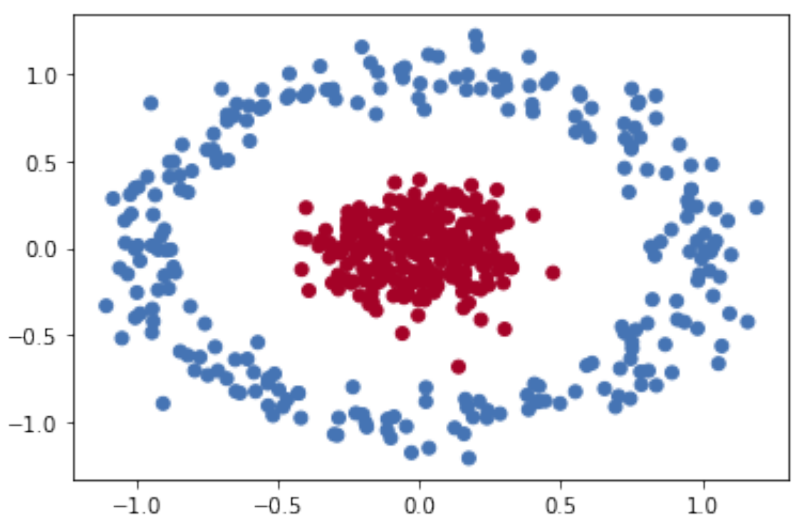
Other Methods for Clustering

In here we will talk about other methods to explore for clustering.

Spectral clustering

Spectral is a more advanced version and as we can see below it does a better job than k means with circle data.

In [5]:
```
cluster = SpectralClustering(n_clusters=2, affinity="nearest_neighbors
labels = cluster.fit_predict(X)  # pass X not similarity matrix
colors=np.array(['#4574B4','#A40227'])
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()
```



Using Random Forest (Breiman's RF for unsupervised learning trick)

We could use random forest as a way to find the distance metric to obervation.

In [ ]: