Library Imports

```
In [1]: import time
        from tqdm import tqdm
        import numpy as np
        import pandas as pd
        from scipy import stats
        from PIL import Image
        import matplotlib.pyplot as plt
        %config InlineBackend.figure_format = 'retina'
        import seaborn as sns
        sns.set(context = 'notebook', style = 'darkgrid')
        import warnings
        warnings.filterwarnings('ignore')

        from sklearn.metrics import confusion_matrix
        from sklearn.neighbors import radius_neighbors_graph
        from scipy.sparse import csgraph

        from sklearn.datasets import load_breast_cancer
        from sklearn.datasets import make_moons
        from sklearn.cluster import SpectralClustering
```

Clustering Algorithms

MSDS 689 HW2 - Joren Libunao

Introduction

Clustering is a popular unsupervised learning technique that you can use to group together datapoints in your dataset based on how similar the datapoints are to each other. There are many practical reasons for why we perform clustering, such as market segmentation, geoanalysis, recommendation engines, etc. In this report we will explore a few of these clustering techniques, how they work, and some of their applications:

- Basic K-means algorithm
- K-means++ algorithm
- Selecting K
- Applications of K-means algorithms
- Spectral clustering

In regards to datasets, we want to use the breast cancer dataset that comes with the Scikit-learn library.

K-means

As mentioned above, K-means is an unsupervised learning algorithm used to cluster datapoints together.

The idea behind the baseline K-means algorithm is to partition the data into a number K clusters, and at the center of each cluster is a centroid, or the mean of the datapoints in the cluster. The algorithm iterates by assigning individual data points to their nearest centroid and updating the centroid to the mean of the data points assigned to it. The algorithm continues to iterate until it converges, or when the centroids are no longer changing significantly.

The step by step process of K-means is listed below:

1. Initialize the centroids: Randomly select K data points as your initial centroids. We will go over choosing K later.
2. Assign each data point to its nearest centroid: Calculate the distance (typically Euclidean) between each data point and the centroids, and assign each data point to the nearest centroid.
3. Update the centroids: Calculate the mean of the data points assigned to each centroid, and update the centroid to be the mean.
4. Repeat steps 2 and 3 until your centroids converge: As the algorithm continues to iterate through these steps, the centroids will move towards convergence and no longer change significantly.

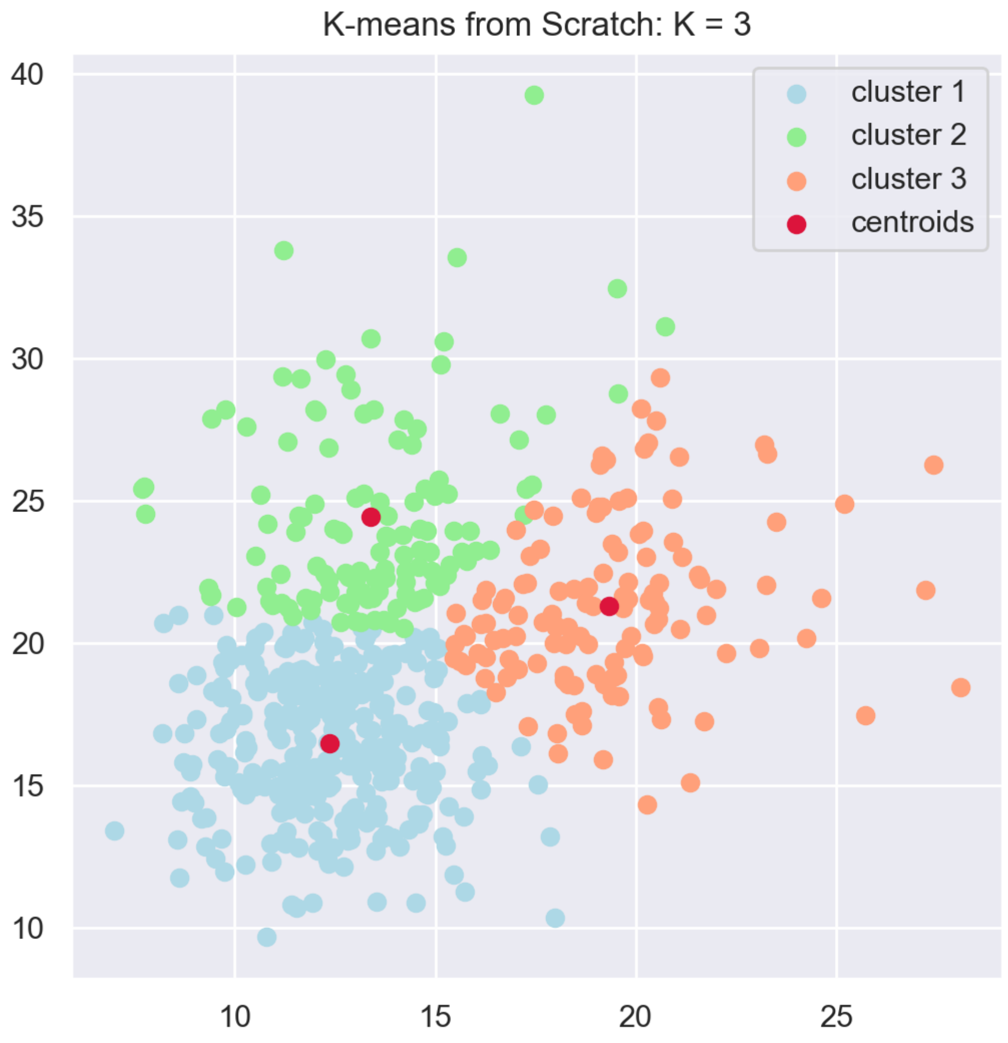
You can set a threshold for what convergence is (ex: when the distance changes by less than 1% compared to the previous iteration). You can also decide to iterate a maximum number of times to save computational time (ex: repeat steps 2 and 3 100 times at most).

Using an implementation built from scratch (which can be found below), we will attempt to cluster the data within the breast cancer dataset from Scikit-learn.

In [2]:
```python
def kmeans(X, k, centroids=None, max_iter=30, tolerance = 1e-2):
    '''
    Basic kmeans algorithm. If centroids = None, then it follows the basic kmea
    algorithm. If centroids = 'kmeans++', then it follows the kmeans++ algorith
    and calls the select_centroids function.
    '''
    # Initialize the centroids randomly (if not already provided)
    if centroids == None:
        centroids = X[np.random.choice(len(X), k, replace=False), :]
    # Assign each data point to its nearest centroid
    for i in range(max_iter):
        prev_centroids = centroids.copy()
        distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum(axis=2))
        labels = np.argmin(distances, axis = 0)
        # Update the centroids
        centroids = np.array([X[labels == j].mean(axis = 0) for j in range(k)])
        # Check for convergence
        if np.linalg.norm(centroids - prev_centroids) / np.linalg.norm(prev_cer
            break
    return centroids, labels
```

Note on visual demonstration: While our K-means algorithm can work with data that is highly dimensional (i.e. more than 2 dimensions), below we are plotting the centroids and original data based on just 2 of the features strictly for visual demonstration purposes. These clusters are not representative of all 30 features since we physically cannot plot 30 dimensions.

In [3]:
```
X, y = load_breast_cancer(return_X_y=True)
centroids, labels = kmeans(X[:,:2], 3, centroids=None, max_iter=100, tolerance=

x_df = pd.DataFrame(X[:,:2])
x_df['labels'] = labels

x0 = x_df[x_df['labels'] == 0]
x1 = x_df[x_df['labels'] == 1]
x2 = x_df[x_df['labels'] == 2]

fig, ax = plt.subplots(figsize = (6,6))
ax.scatter(x0.iloc[:,0], x0.iloc[:,1], c = 'lightblue', label = 'cluster 1')
ax.scatter(x1.iloc[:,0], x1.iloc[:,1], c = 'lightgreen', label = 'cluster 2')
ax.scatter(x2.iloc[:,0], x2.iloc[:,1], c = 'lightsalmon', label = 'cluster 3')
ax.scatter(centroids[:,0], centroids[:,1], c = 'crimson', label = 'centroids')
ax.legend()
ax.set_title('K-means from Scratch: K = 3')
plt.show()
```



```
In [4]: len(x_df[x_df['labels'] == 0]), len(x_df[x_df['labels'] == 1]), len(x_df[x_df['
```

```
Out[4]: (319, 124, 126)
```

We can observe that at K = 3 centroids, the algorithm has clustered the data points into the three differently colored areas. You can also see that the mean of these clusters is roughly where the dark red dots are, which represent the respective centroids of each cluster.

Time Complexity - K-means

The time complexity of this baseline K-means algorithm is $T(n) = ikn$, where

- i = maximum number of iterations
- k = number of clusters
- n = number of observations

Each iteration takes T(n) = kn time, as it needs to compute the distance between each datapoint and each centroid, requiring n times k computations, each of which takes O(1) time. The algorithm assigns each datapoint to a centroid, which takes kn time, and updates the centroids by computing the means of each cluster, which takes kn time as well.

Ultimately, this becomes $T(n) = ikn$. Therefore, the asymptotic complexity of this function would be $O(n)$.

K-means++

K-means++ is a variation of the basic K-means algorithm which improves how the initial centroids are chosen. The general idea is to select initial centroids that are far away from each other to guarantee that they capture different areas of the datapoints. The steps to choosing the initial centroids are as follows:

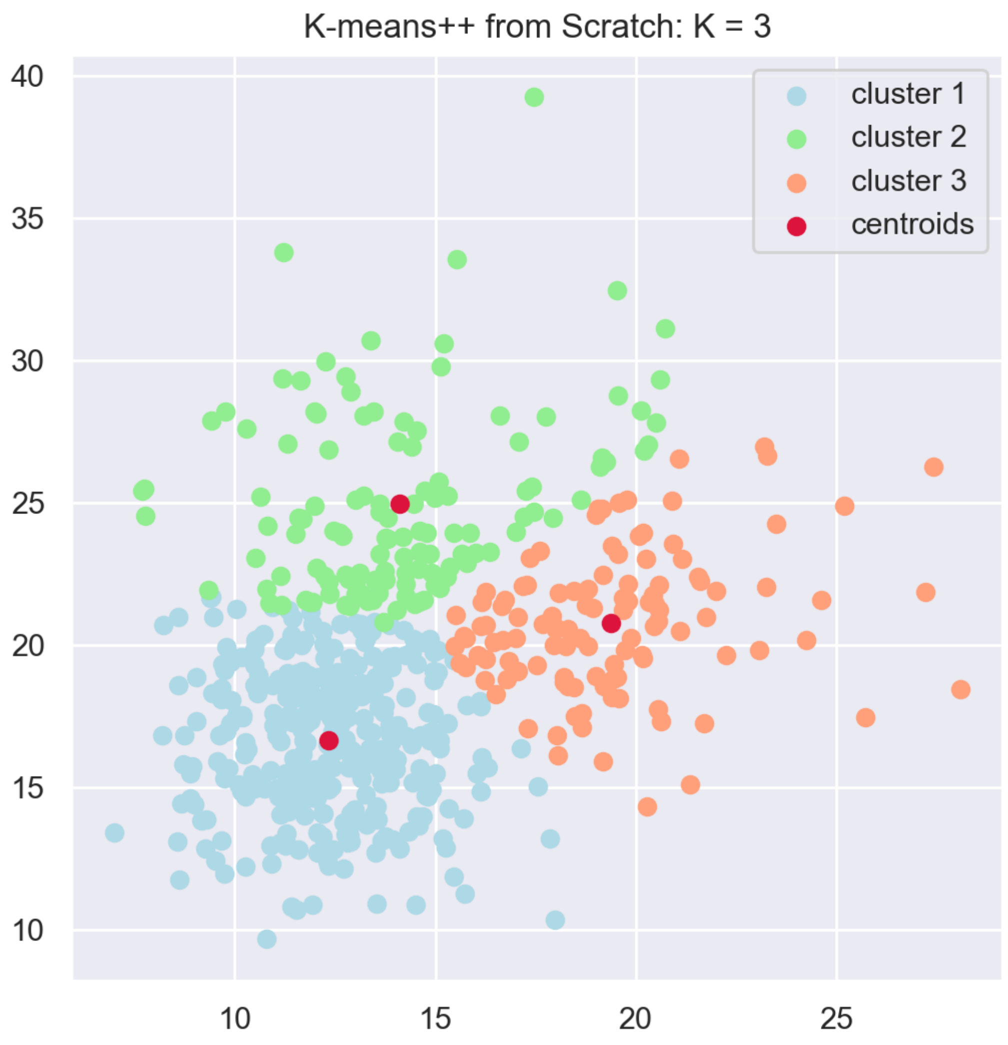
1. Choose the initial centroid randomly from the data points.
2. For each new centroid, calculate the minimum distance between each data point and the closest centroid already chosen. The probability of selecting a data point as the next centroid is proportional to squaring this distance.
3. Repeat step 2 until all k centroids have been chosen.

Below you will find the implementation of K-means we previously used, but revised to include an option specifically for the K-means++ algorithm, simply by specifying `centroids = 'kmeans++'`. We also included a separate function that will select the centroids out of the datapoints based on the K-means++ algorithm.

In [5]:
```
### Function for selecting centroids in kmeans++ implementation
def select_centroids(X, k):
    '''
    kmeans++ algorithm to select initial points:

    1. Pick first point randomly
    2. Pick next k-1 points by selecting points that maximize the minimum
       distance to all existing clusters. So for each point, compute distance
       to each cluster and find that minimum.  Among the min distances to a clu
       for each point, find the max distance. The associated point is the new c

    Return centroids as k x p array of points from X.
    '''
    # Initialize first centroid randomly
    centroids = [X[np.random.choice(len(X), replace=False), :]]
    # Pick next k-1 centroids
    for _ in range(1, k):
        distances = np.array([min([np.linalg.norm(x - centroid) ** 2 for centro
        probs = distances / distances.sum()
        centroids.append(X[np.random.choice(X.shape[0], p=probs)])
    return np.array(centroids)

### K-means algorithm (with K-means++ option)
def kmeans(X, k, centroids=None, max_iter=30, tolerance = 1e-2):
    '''
    Basic kmeans algorithm. If centroids = None, then it follows the basic kmea
    algorithm. If centroids = 'kmeans++', then it follows the kmeans++ algorith
    and calls the select_centroids function.
    '''
    if centroids == None:
        centroids = X[np.random.choice(len(X), k, replace=False), :]
    # If 'kmeans++' is specified, call select_centroids function
    elif centroids == 'kmeans++':
        centroids = select_centroids(X, k)
    for i in range(max_iter):
        prev_centroids = centroids.copy()
        distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum(axis=2))
        labels = np.argmin(distances, axis = 0)
        centroids = np.array([X[labels == j].mean(axis = 0) for j in range(k)])
        if np.linalg.norm(centroids - prev_centroids) / np.linalg.norm(prev_cen
            break
    return centroids, labels
```

Let's use the `'kmeans++'` option to try clustering the datapoints in the breast cancer dataset once again.

In [6]:
```
centroids, labels = kmeans(X[:,:2], 3, centroids='kmeans++', max_iter=100, tole

x_df = pd.DataFrame(X[:,:2])
x_df['labels'] = labels

x0 = x_df[x_df['labels'] == 0]
x1 = x_df[x_df['labels'] == 1]
x2 = x_df[x_df['labels'] == 2]

fig, ax = plt.subplots(figsize = (6,6))
ax.scatter(x0.iloc[:,0], x0.iloc[:,1], c = 'lightblue', label = 'cluster 1')
ax.scatter(x1.iloc[:,0], x1.iloc[:,1], c = 'lightgreen', label = 'cluster 2')
ax.scatter(x2.iloc[:,0], x2.iloc[:,1], c = 'lightsalmon', label = 'cluster 3')
ax.scatter(centroids[:,0], centroids[:,1], c = 'crimson', label = 'centroids')
ax.legend()
ax.set_title('K-means++ from Scratch: K = 3')
plt.show()
```



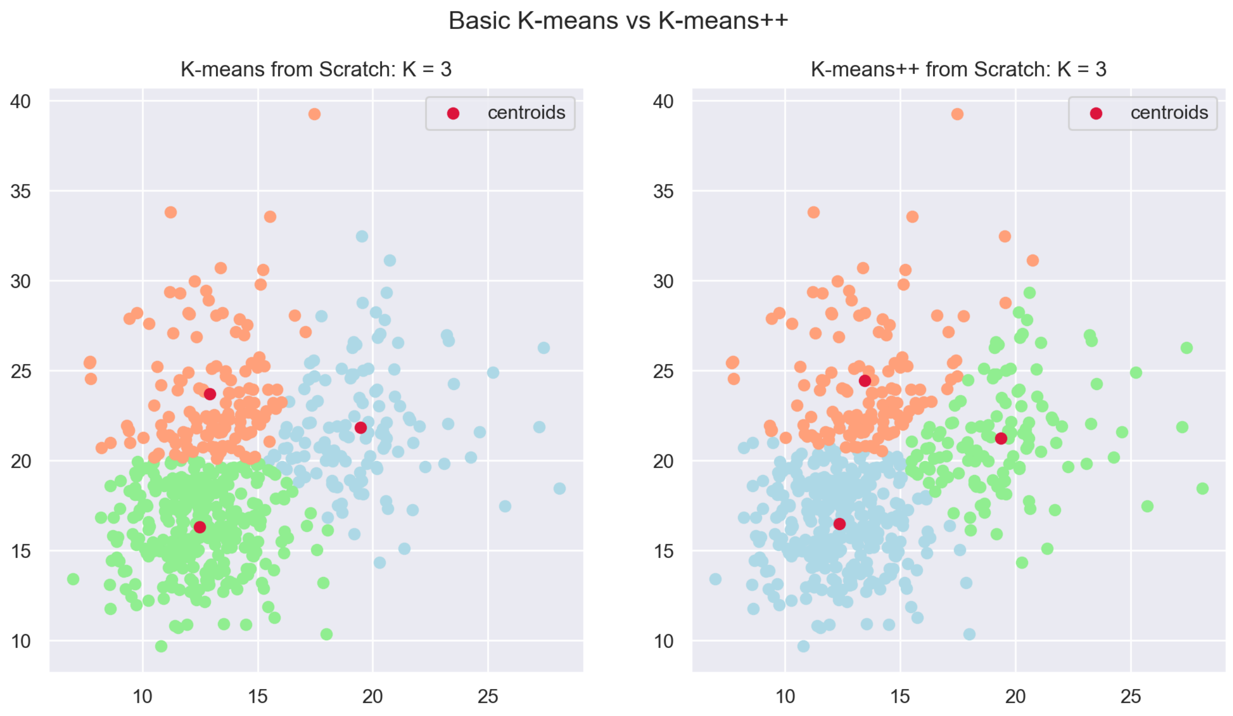
In [7]:
```
len(x_df[x_df['labels'] == 0]), len(x_df[x_df['labels'] == 1]), len(x_df[x_df['
```

Out[7]:
```
(333, 123, 113)
```

Using the K-means++ algorithm option, we can see that the way the datapoints were clustered is slightly different from our original K-means algorithm. To make comparison easier, let's look at their plots side by side.

In [29]:
```
X, y = load_breast_cancer(return_X_y=True)
k_centroids, k_labels = kmeans(X[:,:2], 3, centroids=None, max_iter=100, tolera
kpp_centroids, kpp_labels = kmeans(X[:,:2], 3, centroids='kmeans++', max_iter=1

x_df = pd.DataFrame(X[:,:2])
x_df['k_labels'] = k_labels
x_df['kpp_labels'] = kpp_labels

kx0 = x_df[x_df['k_labels'] == 0]
kx1 = x_df[x_df['k_labels'] == 1]
kx2 = x_df[x_df['k_labels'] == 2]
kppx0 = x_df[x_df['kpp_labels'] == 0]
kppx1 = x_df[x_df['kpp_labels'] == 1]
kppx2 = x_df[x_df['kpp_labels'] == 2]

fig, ax = plt.subplots(1,2,figsize = (12,6))
```

```
ax = ax.flatten()

ax[0].scatter(kx0.iloc[:,0], kx0.iloc[:,1], c = 'lightblue')
ax[0].scatter(kx1.iloc[:,0], kx1.iloc[:,1], c = 'lightgreen')
ax[0].scatter(kx2.iloc[:,0], kx2.iloc[:,1], c = 'lightsalmon')
ax[0].scatter(k_centroids[:,0], k_centroids[:,1], c = 'crimson', label = 'centr
ax[0].legend()
ax[0].set_title('K-means from Scratch: K = 3')

ax[1].scatter(kppx0.iloc[:,0], kppx0.iloc[:,1], c = 'lightblue')
ax[1].scatter(kppx1.iloc[:,0], kppx1.iloc[:,1], c = 'lightsalmon')
ax[1].scatter(kppx2.iloc[:,0], kppx2.iloc[:,1], c = 'lightgreen')
ax[1].scatter(kpp_centroids[:,0], kpp_centroids[:,1], c = 'crimson', label = 'c
ax[1].legend()
ax[1].set_title('K-means++ from Scratch: K = 3')

fig.suptitle('Basic K-means vs K-means++')
plt.show()
```



While the change is hard to notice, the ultimate goal of each algorithm is essentially the same, and so it is only natural that they end up in similar clusters. Again, the only difference between the two is how the centroids are initialized. If we were to reduce `max_iter` to a much lower number, we would probably see the difference immediately as it would not have as many iterations to get to the same result.

Time Complexity - K-means++

The only difference between the time complexities of K-means++ and the original K-means algorithm is the initialization of the centroids, which takes nk time as it must iteratively select centroids based on the first randomly selected centroid, and to do that it must calculate the distance between each datapoint and the nearest centroid. Therefore, $T(n) = ikn + kn$, and the asymptotic complexity is $O(n)$.

Comparing Performance Speeds

Let's test our time complexity theories with some good old fashioned speed testing. We will create some sample data and see how fast each version of the algorithm can cluster the data. Because of the random aspect of choosing centroids in each case, we can't just run the functions once and leave it up to chance. We will run each function 1000 times and take the average to get a more stabilized estimate for the speed of each algorithm.

First let's make some sample data about test grades:

In [9]:
```
grades = [92.65, 93.87, 74.06, 86.94, 92.26, 94.46, 92.94, 80.65, 92.86,
          85.94, 91.79, 95.23, 85.37, 87.85, 97.71, 93.03]
grades = np.array(grades).reshape(-1,1)
k = 3 # Representing A, B, and C grades
```

Now, let's get the mean performance speed of the baseline K-means algorithm:

In [10]:
```
kmeans_times = []
for _ in tqdm(range(1000)):
    start_time = time.time()
    centroids, labels = kmeans(grades, k)
    kmeans_times.append(time.time() - start_time)
print(f'{np.mean(kmeans_times)} seconds')
```

```
100%|██████████████████████████████████████| 1000/1000 [00:00<00:00, 3131.96it/
s]
0.0003139655590057373 seconds
```

And now for the mean performance speed of the K-means++ algorithm:

In [11]:
```
kmeanspp_times = []
for _ in tqdm(range(1000)):
    start_time = time.time()
    centroids, labels = kmeans(grades, k, centroids = 'kmeans++')
    kmeanspp_times.append(time.time() - start_time)
print(f'{np.mean(kmeanspp_times)} seconds')
```

```
100%|██████████████████████████████████████| 1000/1000 [00:00<00:00, 1320.90it/
s]
0.0007478425502777l seconds
```

The difference is very marginal, but nonetheless it exists. It is clear that the K-means++ algorithm is more than twice as slow by a few ten-thousandths of a second. What if we tried this on the breast cancer dataset that we previously used?

In [12]:
```
X, y = load_breast_cancer(return_X_y=True)
```

In [13]:
```
kmeans_times = []
for _ in tqdm(range(1000)):
    start_time = time.time()
    centroids, labels = kmeans(X, k)
    kmeans_times.append(time.time() - start_time)
print(f'{np.mean(kmeans_times)} seconds')
```

```
100%|██████████████████████████████████████| 1000/1000 [00:02<00:00, 348.54it/
s]
0.0028415746688842773 seconds
```

In [14]:

```
kmeanspp_times = []
for _ in tqdm(range(1000)):
    start_time = time.time()
    centroids, labels = kmeans(X, k, centroids = 'kmeans++')
    kmeanspp_times.append(time.time() - start_time)
print(f'{np.mean(kmeanspp_times)} seconds')
```

```
100%|███████████████████████████████████████| 1000/1000 [00:11<00:00, 87.50it/
s]
0.011326839447021484 seconds
```

The difference in speeds is a bit more pronounced. The baseline algorithm takes only thousandths of a second to cluster the data, while the K-means++ algorithm takes hundredths of a second to do the same. The bottom line is that the K-means++ algorithm is slower at clustering due to the initialization of the centroids.

Selecting k

K-means is a great algorithm for clustering datapoints, but how do we choose what `k` is? In certain business cases, you would want to select your `k` based on domain knowledge. Going back to our breast cancer example, you would pick `k = 2` as the patient can be either malignant or benign, and there are not really any other options besides those two.

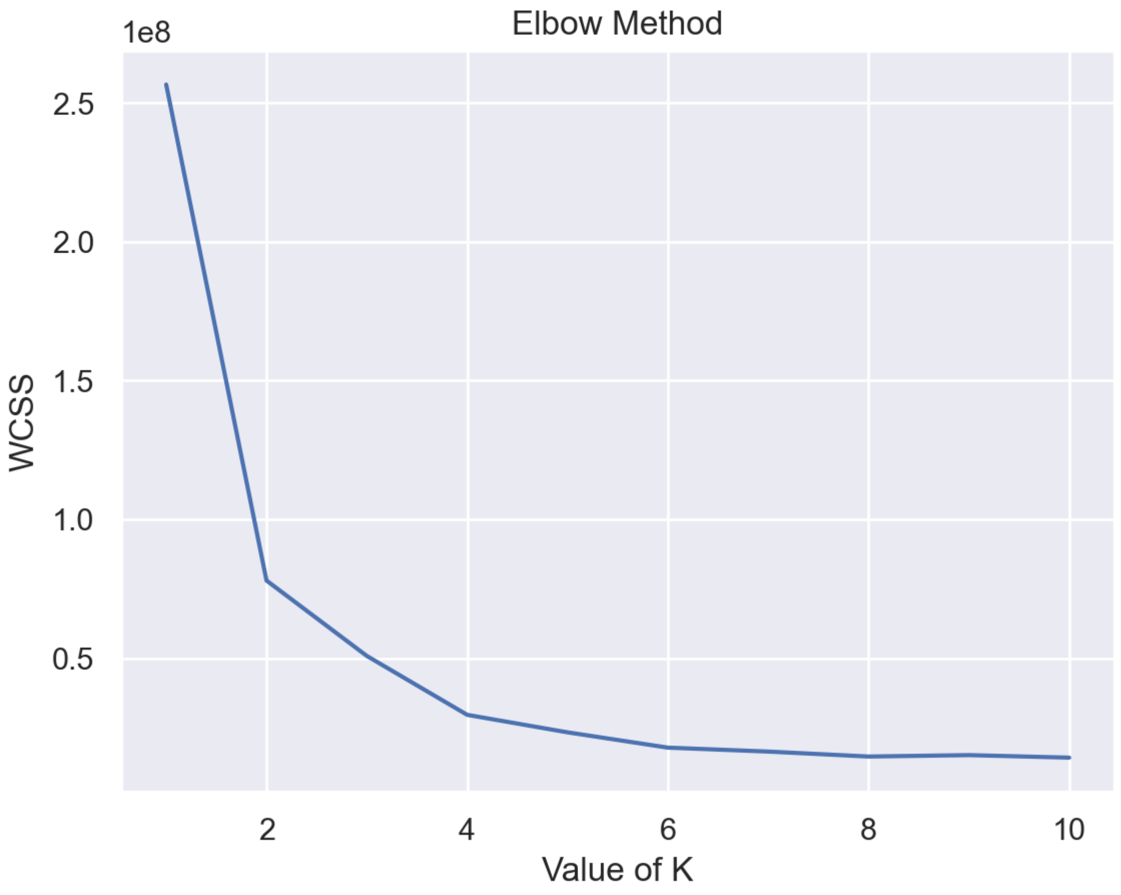
Elbow Method

What about cases where we don't know how many clusters we need or want? We can use something called the *"Elbow method"* to determine the ideal value for `k`. The idea behind the elbow method is plotting different values of `k` and their respective *Within Cluster Sum of Squares*, or WCSS. This metric represents the squared distances between the datapoints and the centroids, which we calculated within our K-means implementation. We can take different values of `k`, calculate their respective WCSS, and then plot these values on a graph, like below.

In [15]:

```
def plot_elbow(X, max_k):
    '''
    With respect to the K-means algorithm, plots the Elbow method for the giver
    dataset and maximum K value.
    '''
    wcss_ = []
    for k in range(1, max_k + 1):
        centroids, labels = kmeans(X, k)
        wcss = 0
        # For each centroid, calculate WCSS
        for k_ in range(k):
            idx = labels == k_
            if np.sum(idx) > 0:
                dist = np.sum((X[idx,:] - centroids[k_,:]) ** 2)
                wcss += dist
        wcss_.append(wcss)
    # Plot the elbow method line
    plt.plot(range(1, max_k+1), wcss_)
    plt.title('Elbow Method')
    plt.xlabel('Value of K')
    plt.ylabel('WCSS')
    plt.show()
```

In [16]:
```
plot_elbow(X, 10)
```



Where does the "elbow" part come from? If you look at the plot above, you can see that there is a value K = 2 where the WCSS drops significantly compared to K - 1 = 1. This is considered the "elbow" due to how the plot resembles an arm. Every K after the elbow, the WCSS diminishes very marginally, and at that point you are just creating smaller, less meaningful clusters. To make your clustering more meaningful, we pick the K that minimizes WCSS while also avoiding diminishing returns from picking larger values of K.

Silhouette Method

There is also an alternate method to choose K called the *"Silhouette Method"*. In this method, the idea is to measure how similar a datapoint is to other datapoints in the same cluster compared to datapoints in the next nearest cluster. The formula for calculating an observation's silhouette score is

$$S_i = \frac{b_i - a_i}{max\{a_i, b_i\}}$$

where b_i represents the average distance between datapoint i and all points in the next nearest cluster and a_i represents the average distance between datapoint i and all points in the same cluster. The value S_i can range from -1 to +1, and ideally you want it to be close to +1. For each cluster k, you would take the mean of the silhouette scores ($S_{i,k}$) of all the datapoints, and then choose the value of k that maximizes the mean. In mathematical terms, this looks like

$\max_k \bar{S}_k$, where $\bar{S}_k = \frac{1}{n}\Sigma_{i=1}^{n} S_{i,k}$

Applications of K-Means Algorithms

Breast Cancer Predictions

We can use K-means algorithms to do some classification as well. If we take the breast cancer dataset above and classify `y = 0` as meaning the person does not have cancer and `y = 1` means they do have cancer, we can use clustering to make predictions on whether the person has cancer or not. We can assign a label of 0 or 1 to a cluster depending on which prediction is the majority among the datapoints within the cluster.

There is a small problem with this method, however, as it is unclear which centroid belongs to which label, since we do not pass the target data to the K-means algorithm. To ensure the labels match up, we can take the most common prediction of the datapoints within one cluster and assume that is the prediction, and then flip each element in that cluster to the appropriate label. These new labels can then be compared to the ground truth `y`.

In [17]:
```
def kmeans_confusion_matrix(y, labels):
    corrected_labels = np.zeros(len(labels))
    for label in np.unique(labels):
        if stats.mode(y[labels == label]).mode[0] != label:
            corrected_labels[labels != label] = label
        else:
            corrected_labels[labels == label] = label
    return confusion_matrix(y, corrected_labels)
```

In [18]:
```
X, y = load_breast_cancer(return_X_y=True)
centroids, labels = kmeans(X, 2, centroids='kmeans++')
kmeans_confusion_matrix(y, labels)
```

Out[18]:
```
array([[128,  84],
       [  1, 356]])
```

If you are not familiar with confusion matrices, the top left and bottom right values represent the "true positive" and "true negative" observations, respectively. These observations were accurately predicted by the K-means algorithm. The top right and bottom left values represent the "false positive" and "false negative" observations, respectively. These observations were inaccurately predicted by the K-means algorithm. As you can see, most of the values were predicted correctly. In order to improve the accuracy further, we may have to use more advanced clustering techniques.

Image Compression

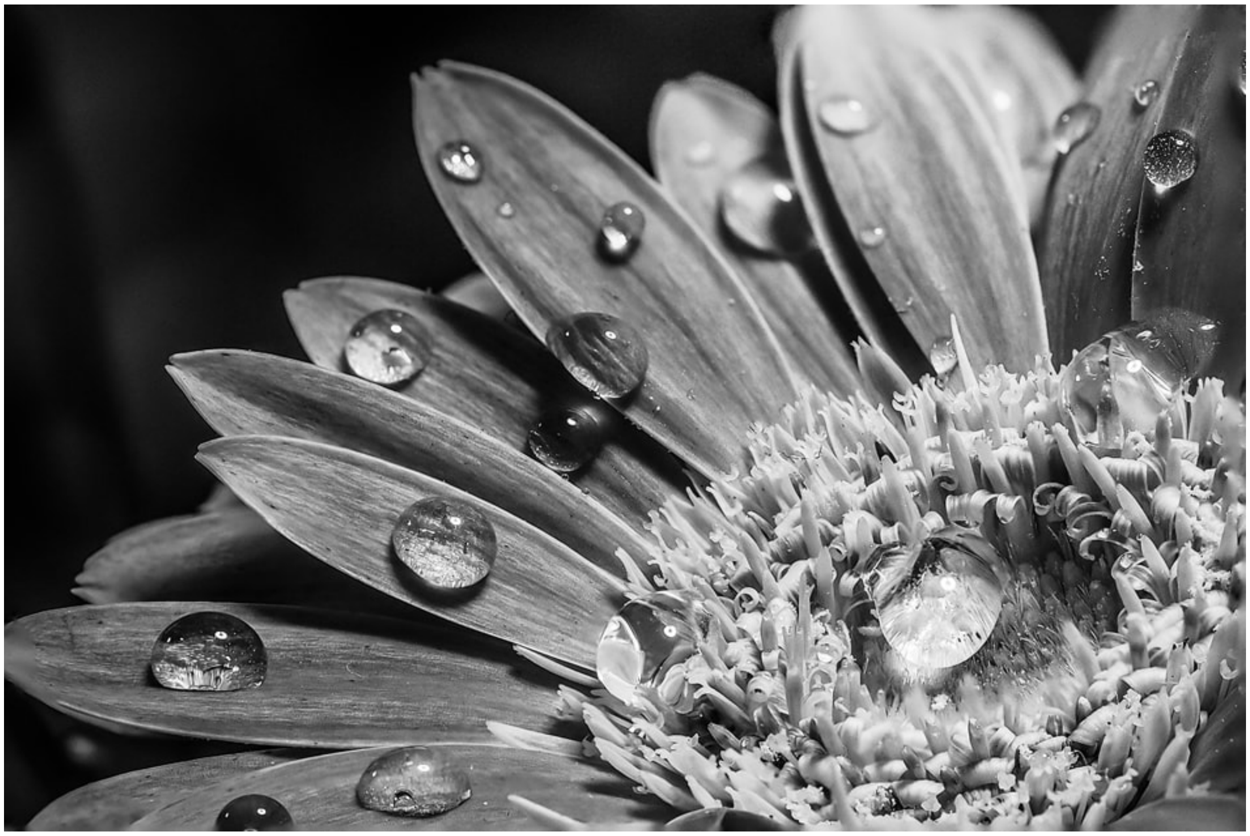
We can also use K-means clustering to compress an image. The general idea is to take the raw RGB values of the pixels in the image and cluster them into `k` clusters, or colors. Once they are clustered, we replace each pixel's RGB values with the RGB values of the cluster's centroid, thus creating a newly compressed image.

In [19]:
```
def kmeans_image(image_path, k):
    '''
    Compresses an image using k-means clustering on the RGB values of its pixel
    Returns a compressed PIL image object.
    '''
    image = Image.open(image_path)
    X = np.array(image).reshape(-1, 3)
    centroids, labels = kmeans(X, k=k, centroids='kmeans++')
    compressed_X = np.array([centroids[l] for l in labels])
    compressed_image = Image.fromarray(compressed_X.reshape(image.size[1], imag
    return compressed_image
```

> To start simple, let's try it on a black and white image. Here is the original:

In [20]:
```
Image.open('flower.jpeg')
```

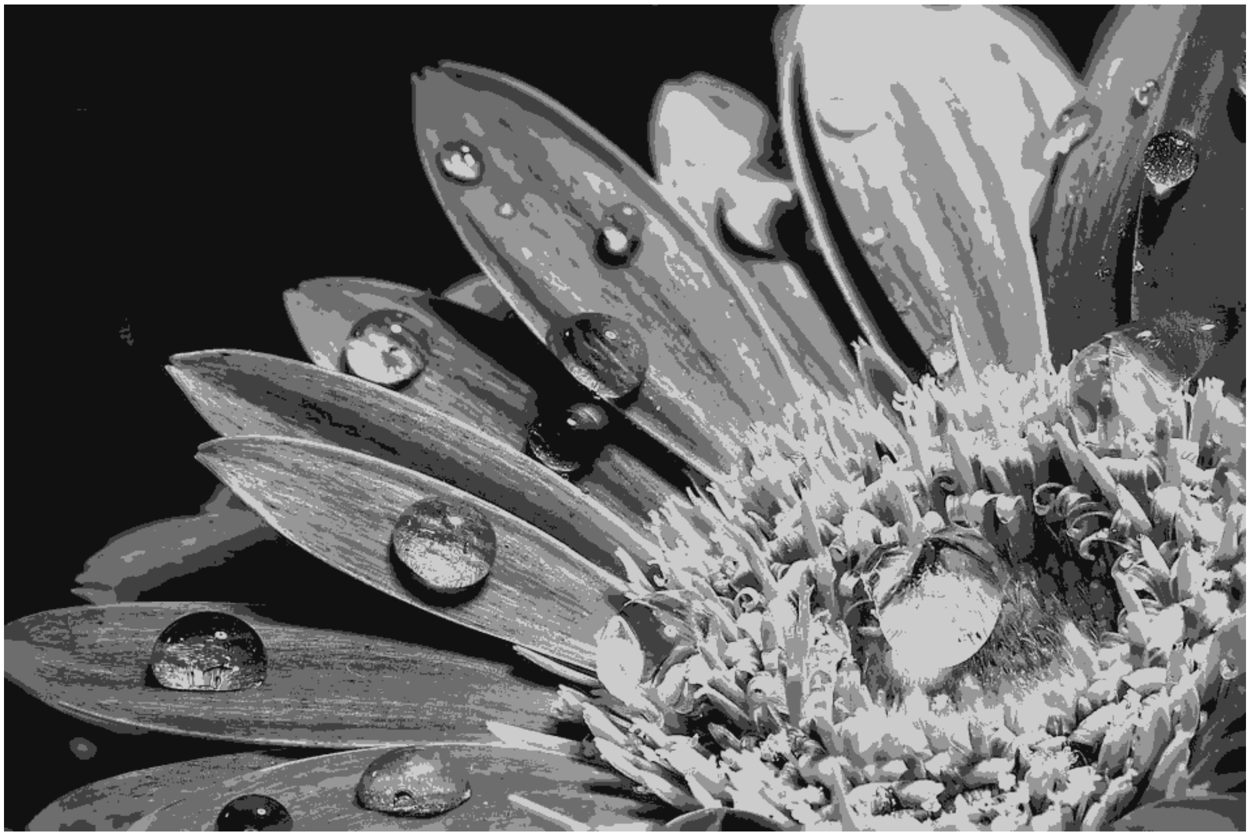
Out[20]:



> And after using K-means clustering, here is the compressed image with `k = 5`:

In [21]:
```
kmeans_image('flower.jpeg', 5)
```

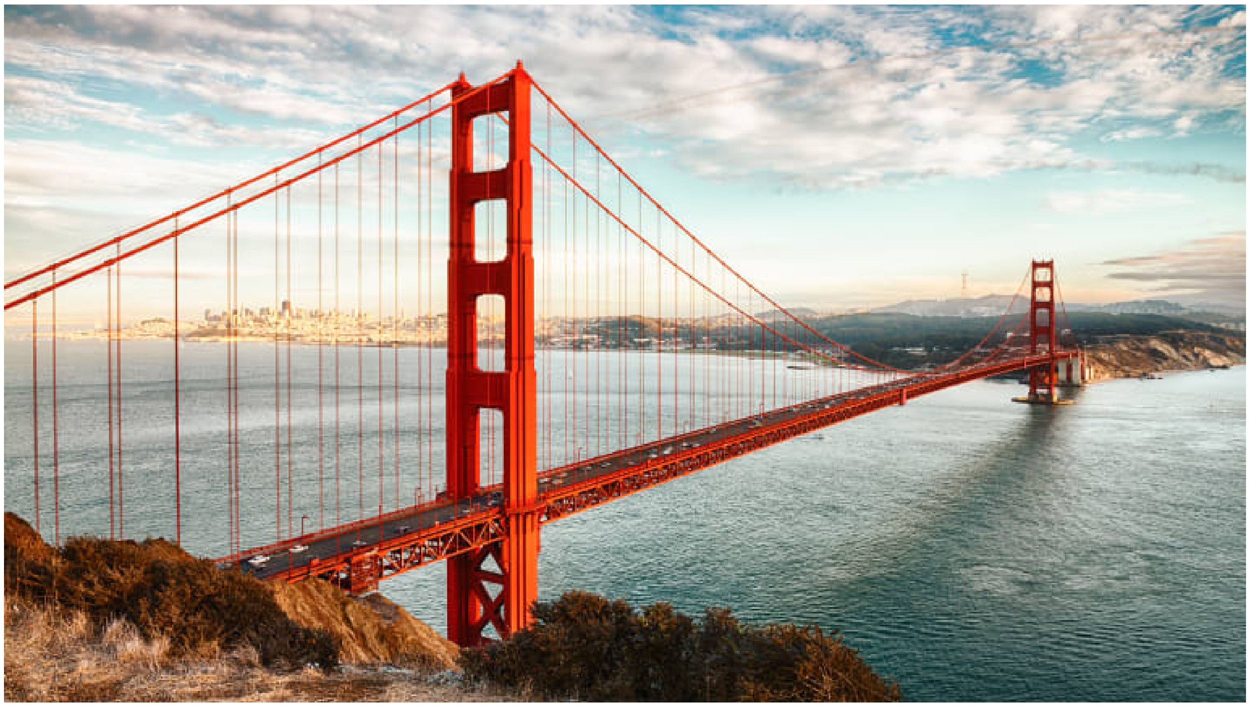
Out[21]:



> K-means did a relatively good job of recreating the photo with fewer colors on the black-white spectrum.
>
> Now let's try doing it with a color photo. Here is a color photo of the Golden Gate bridge:

In [22]:
```
Image.open('bridge.jpeg')
```

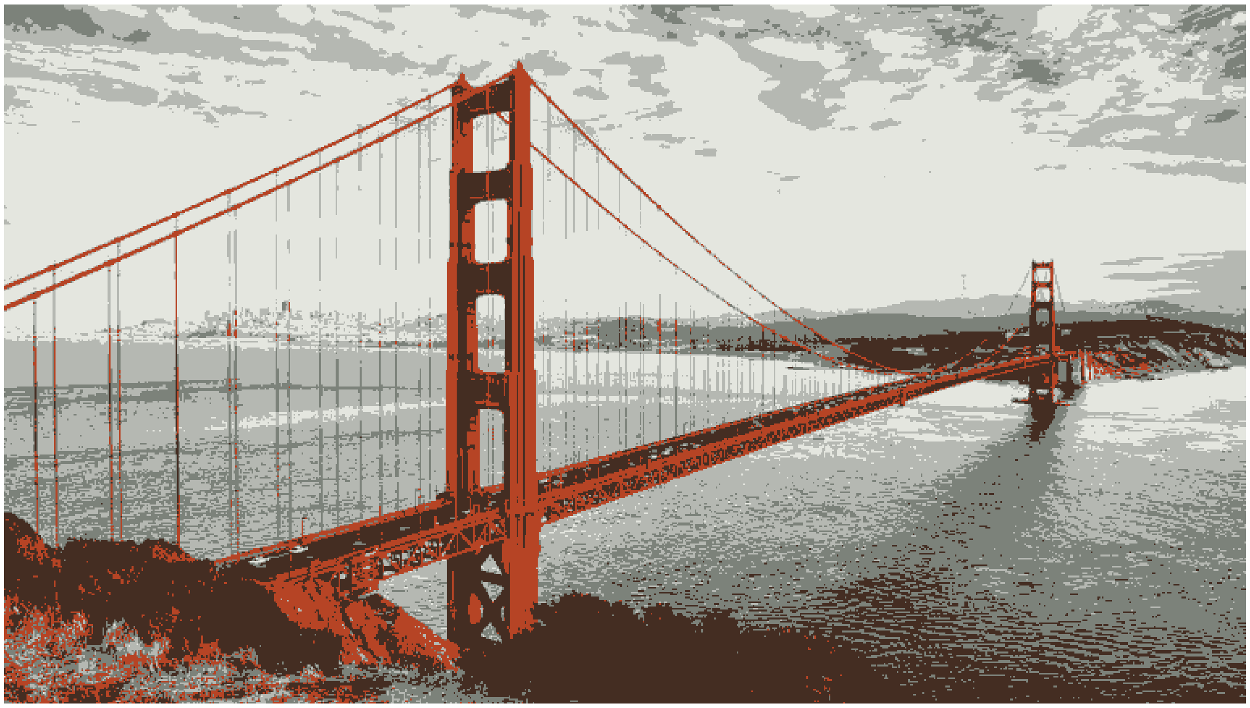
Out[22]:



> And now here it is again using K-means clustering to compress the image with `k = 5` again:

In [23]:
```
kmeans_image('bridge.jpeg', 5)
```

Out[23]:



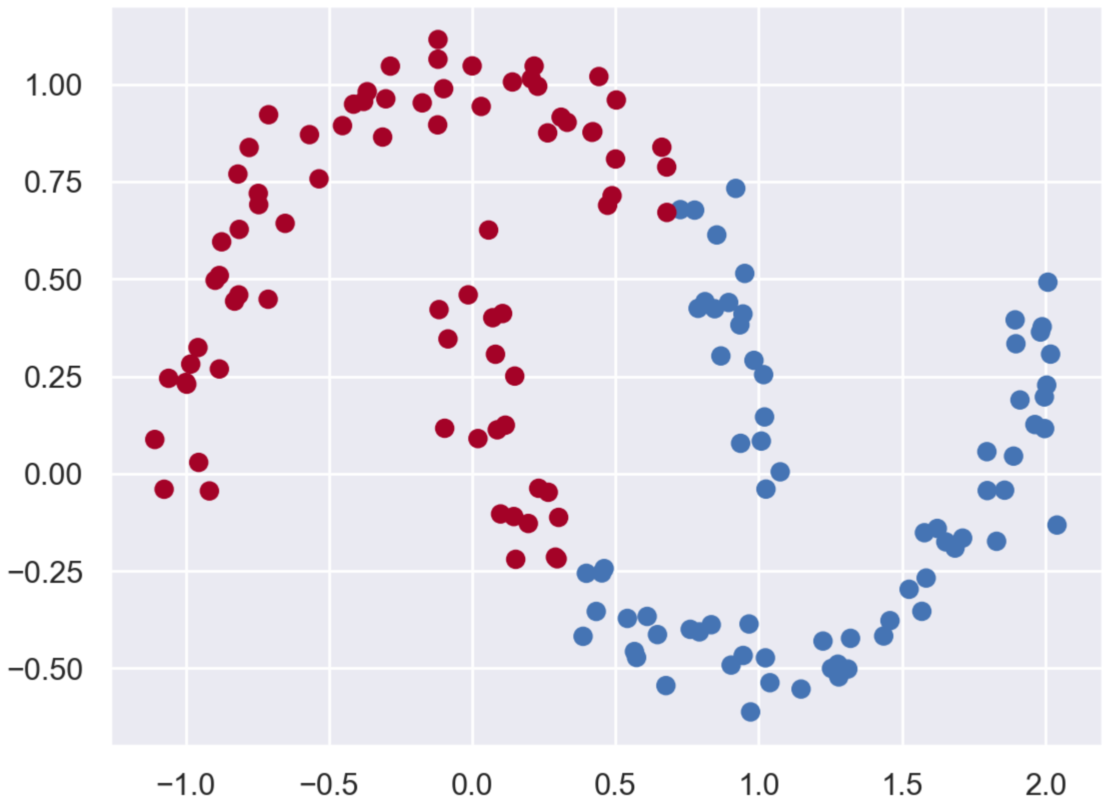
Spectral Clustering

K-means is a useful and simple to explain clustering algorithm, but it does not always work as intended. Let's look at an example where K-means doesn't work. Here is synthetic data from two distributions that are like convex curves.

In [24]:

```
X, _ = make_moons(150, noise=0.07, random_state=21)
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()
```

```
[[ 1.21728954 -0.0628205 ]
 [-0.19634299  0.55201927]]
```



As you can see above, K-means performs poorly on non-linear, convex clusters. It still divided the data into two clusters, but not as intended. Visually, we can distinguish that there are two clusters with parabolic shapes, one pointing downward and one pointing upward.

In these cases, we may have to turn to more advanced clustering techniques, like spectral clustering. Spectral clustering is another type of unsupervised learning / clustering technique that can cluster the data much better in certain edge cases that the original K-means algorithm cannot, at least by itself.

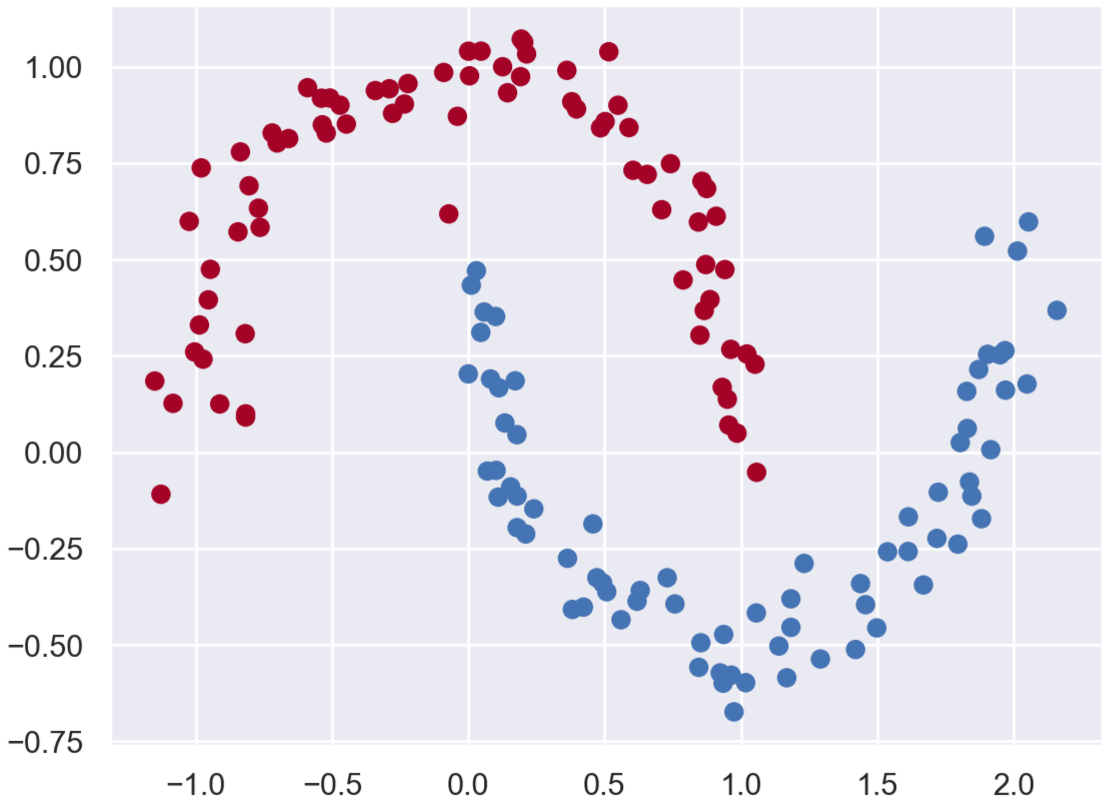
There is a significant amount of matrix calculations and linear algebra that are part of its process. To keep it as simple as possible, the general idea behind spectral clustering is to represent the datapoints as nodes in a graph, where the edges, or relationships, between nodes represent the similarity between the datapoints. If there is no edge between two points, that means they are dissimilar. To cluster the datapoints, spectral clustering constructs a graph based on the similarity between pairs of datapoints, and then finds the "eigenvectors" of the graph Laplacian matrix, which is a matrix that contains information on the graph's structure. These eigenvectors are then used to transform the data into a new space where clustering is feasible.

This is fairly difficult to implement completely from scratch, so I have built one using `radius_neighbors_graph` from `sklearn.neighbors` and `csgraph` from `scipy.sparse`.

In [25]:
```
def spectral_clustering(X):
    '''
    Spectral clustering algorithm using sklearn.neighbors radius_neighbors_grap
    scipy.sparse csgraph.
    '''
    # Build the affinity matrix
    A = radius_neighbors_graph(X, 0.4, mode='distance', metric='minkowski',
                               p=2, metric_params=None, include_self=False).toa
    # Calculate the Laplacian matrix
    L = csgraph.laplacian(A, normed=False)
    # Get labels from eigenvectors of Laplacian matrix
    eigval, eigvec = np.linalg.eig(L)
    labels = eigvec[:,1].copy()
    labels[labels < 0] = 0
    labels[labels > 0] = 1
    return labels
```

Let's see how the scratch implementation performs on the two parabolic distributions from above.

In [35]:
```
X, _ = make_moons(150, noise=0.07)
labels = spectral_clustering(X)
labels = [int(label) for label in labels]

colors=np.array(['#4574B4','#A40227'])
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()
```



As you can see above, spectral clustering was able to almost perfectly identify the two parabolic distributions as separate clusters, unlike K-means. Spectral clustering does an incredible job at handling data that is difficult to cluster using traditional clustering algorithms, as it can handle very complex cluster shapes such as these.

Time Complexity - Spectral Clustering

You may wonder, why not just use spectral clustering all the time if it's so good? Well, depending on your dataset, the time complexity of spectral clustering can be rather long. The time complexity can vary depending on the exact method used, but this research paper by Donghui Yan, Ling Huang, and Michael I. Jordan, researchers at the University of California, Berkeley and the Intel Research Lab, indicates that the asymptotic complexity of spectral clustering in general is $O(n^3)$. The paper states that spectral clustering does not compete well with "classical algorithms such as hierarchical clustering and k-means for large-scale data mining problems" due to the fact that the process involves constructing an $n * n$ affinity matrix and then computing the eigenvectors of this matrix. As a result, larger datasets become impossible to use spectral clustering on, making it impractical compared to K-means.

Performance Speeds

Let's see how fast spectral clustering performs in comparison to our original K-means algorithms. We will use the fake `grades` sub-dataset from our initial speed tests of the K-means algorithms to keep it simple, as we are expecting this to take much longer so it is infeasible to use it on the breast cancer dataset.

In [27]:
```
sc_times = []
for _ in tqdm(range(1000)):
    start_time = time.time()
    labels = spectral_clustering(grades)
    sc_times.append(time.time() - start_time)
print(f'{np.mean(sc_times)} seconds')
```

```
100%|████████████████████████████████████████| 1000/1000 [00:01<00:00, 889.51it/
s]
0.0011106204986572265 seconds
```

As expected, the performance speed for spectral clustering was slower than K-means or K-means++ due to its high asymptotic complexity. With an average performance time of 0.001 seconds, this was four ten-thousandth seconds slower than K-means++ on the same `grades` sub-dataset. So as we demonstrated, yes it can handle complicated cluster shapes, but it will run slower than traditional clustering techniques.

Conducting the same test using Scikit-learn's implementation for spectral clustering takes significantly longer at about 0.01 seconds, as that function is more robust to varying cluster shapes than our basic implementation built from scratch, but as a result is computationally expensive.

In [28]:
```
sc_times = []
for _ in tqdm(range(1000)):
    start_time = time.time()
    cluster = SpectralClustering(n_clusters=2, affinity='nearest_neighbors', as
    labels = cluster.fit_predict(grades)
    sc_times.append(time.time() - start_time)
print(f'{np.mean(sc_times)} seconds')
```

```
100%|██████████████████████████████████████| 1000/1000 [00:13<00:00, 75.11it/
s]
0.013166147708892822 seconds
```

Conclusion

K-means stands as one of the most popular clustering techniques due to its simplicity, as well as its flexibility with nearly any dataset. While it does have its limitations as shown in the Spectral Clustering section, most datasets will not have distributions like that, and thus, K-means remains universally applicable. However, it is important to keep in mind that there is unfortunately no ultimate clustering technique to end all techniques--it is best to choose a clustering technique depending on your given dataset and problem.

There are many more clustering techniques that can be explored which handle various kinds of data, such as DBSCAN, or Density-Based Spatial Clustering of Applications with Noise. This clustering technique can handle datasets with clusters that have similar densities but can also simultaneously identify datapoints as noise. Scikit-learn's `sklearn.cluster` module comes with this and many other clustering techniques which are worth exploring if you need something a bit more advanced than the basic K-means algorithm.