Image Segmentation Using Clustering

SRIHARSHA GADDIPATI

CLUSTERING - UNSUPERVISED LEARNING

clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters).

Types of clustering

1. Connectivity Models - Hierarchical Clustering

(Agglomerative)

2. Centroid Models - KMeans

3. Distribution Models - Expectation-maximization algo

4. Density Models - DBSCAN

Now for image segmentation we will apply the KMeans, MeanShift, Agglomerative, DBSCAN, Gaussian Mixture Model(GMM) and see how they works

Getting input ready

- First load the image using opencv library(imread) to your program
- The shape of your image array(H,W,3) will have 3 dimensions (i.e. each dimension for storing R,G,B values of a pixel)
- Flatten to your image array to (H*W,3)
- Now you can use this array as your input to clustering algorithm

```
img = cv2.imread("./image/35010.jpg",1)

image = img.reshape(-1, img.shape[-1])
```

KMEANS

The algorithm works iteratively to assign each data point to one of *K* groups based on the features that are provided. Data points are clustered based on feature similarity. The results of the *K*-means clustering algorithm are:

1. The centroids of the *K* clusters, which can be used to label new data
2. Labels for the training data (each data point is assigned to a single cluster)

Procedure for Kmeans

1. Place K points into space represented by the objects that are being clustered. These points represent initial group centroids.
2. Assign each pixel to the group that has the closest centroid.
3. When all pixels have been assigned, recalculate the positions of the K centroids
4. Repeat steps 2 and 3 until the centroids no longer move. This produces a seperation of the pixels into groups from which the metric to be minimized can be computed.

Objective Function

KMeans clusters the M points into K clusters by minimizing the squared error function

$$\sum_{i=1}^{k} \sum_{x_j \in S_i} \left(x_j - \mu_i\right)^2$$

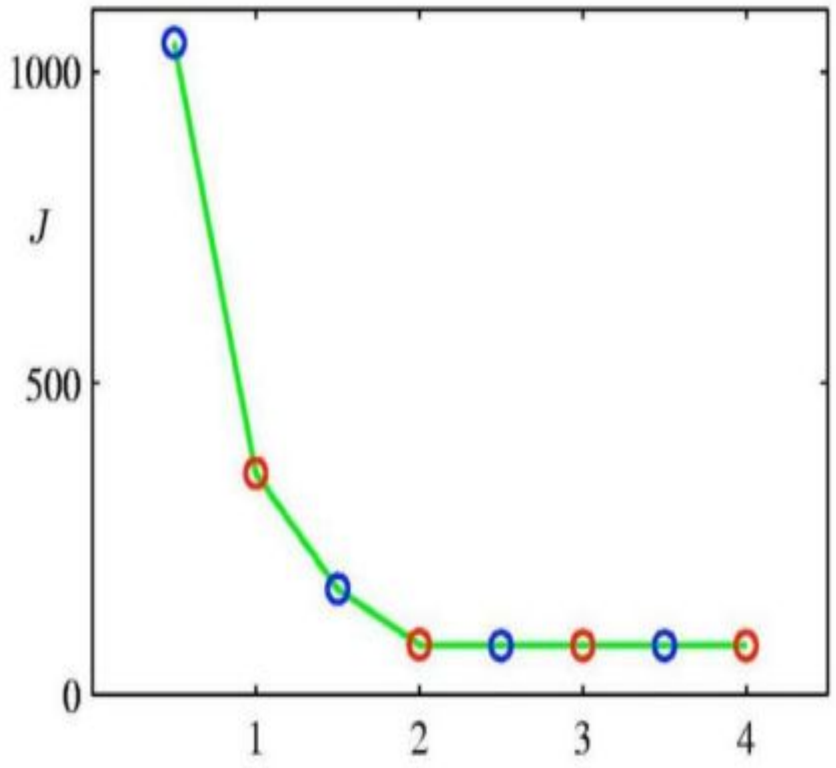
Clusters Si i=1,2...k

Ui is the centroid of all Xj that belongs to Si

Choosing K

One way to select K for the KMeans is to try different values of k, plot the kmeans objective versus K and look at the elbow point in the plot.

For side plot, K=2 is the elbow point

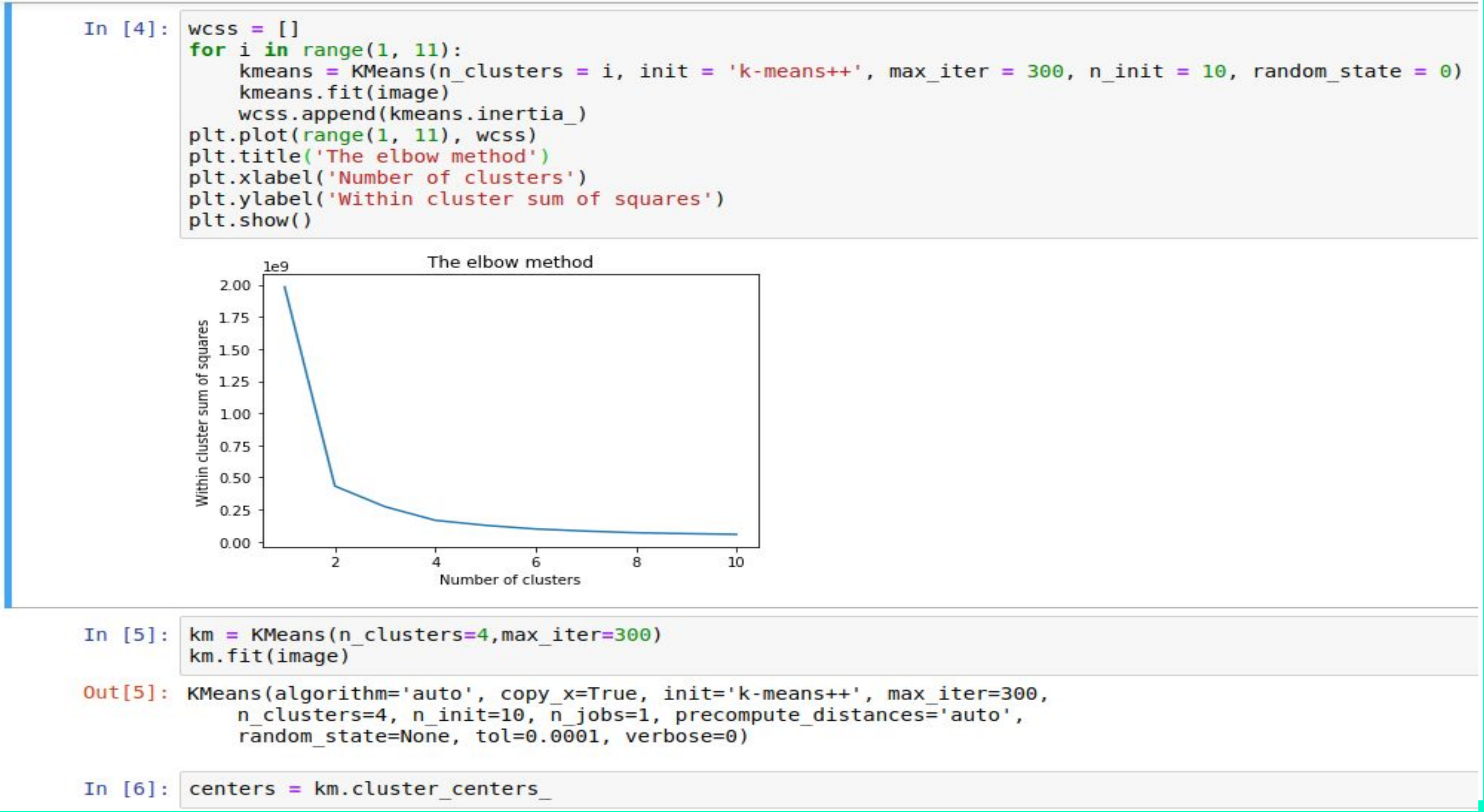


Pros

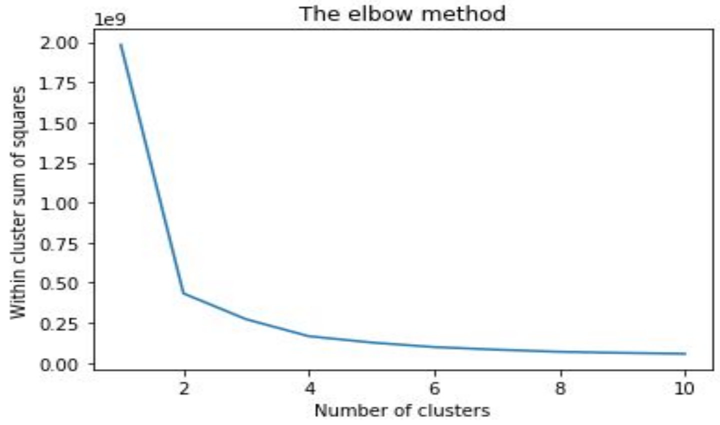
- With a large number of variables, KMeans may be computationally faster than hierarchical clustering
- KMeans produce lighter clusters than hierarchical clustering, especially if the clusters are globular
- The first most used clustering algorithm

Cons

- Different initial partitions can result in different final clusters
- K-Means often doesn’t work when clusters are not round shaped



In [4]:
```
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 300, n_init = 10, random_state = 0)
    kmeans.fit(image)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Within cluster sum of squares')
plt.show()
```

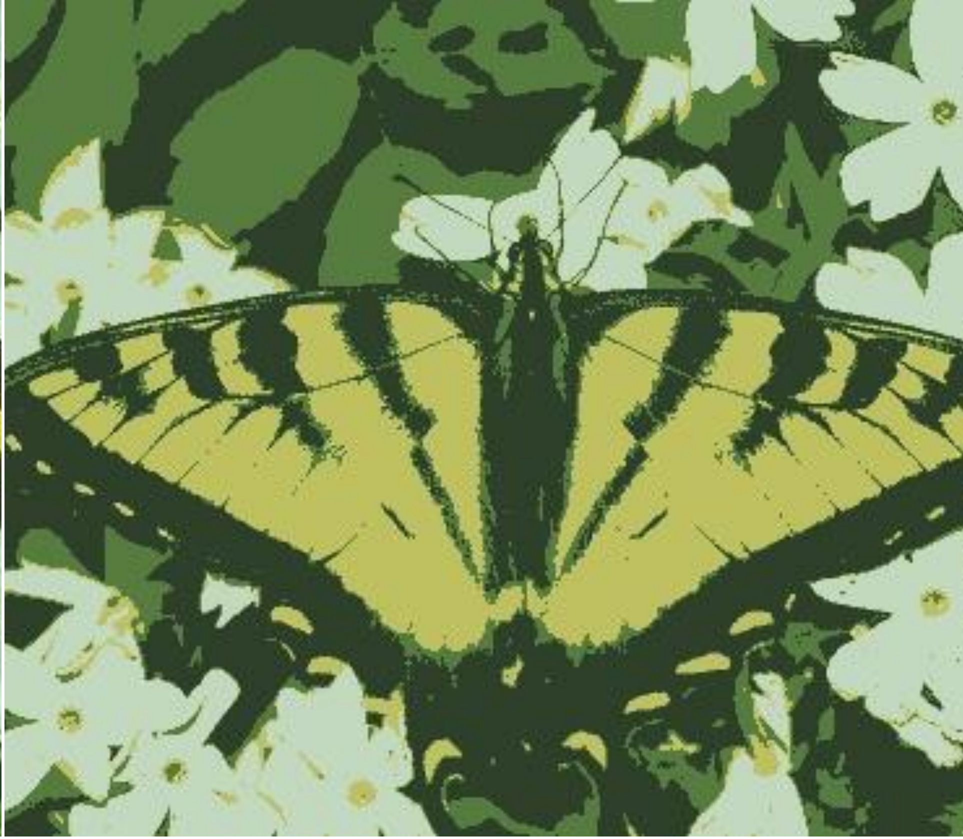


In [5]:
```
km = KMeans(n_clusters=4,max_iter=300)
km.fit(image)
```

Out[5]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=4, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

In [6]:
```
centers = km.cluster_centers_
```

Before Kmeans



After Kmeans

DBSCAN

***D*ensity-*B*ased *S*patial *C*lustering of *A*pplications with *N*oise**
is a well known data clustering algorithm

Based on a set of points, DBSCAN groups together points that are close to each other based on distance measurement and a minimum number of points. It also marks as outliers the points that are in low density regions

Parameters

EPS

The minimum distance between two points. It means that if the distance two points is lower or equal to this value(eps), these points are considered as neighbours

MinPoints

It The minimum number of points to form a dense region. For example, if we set the minpoints parameter as 5, then we need at least 5 points to form a dense region

Types of points

Core Point

A point is a core point if it has more than a specified number of points (MinPoints) within Eps. These are points that are at the interior of a cluster

Border Point

A border point has fewer than MinPoints within Eps, but is in the neighborhood of a core point

Noise Point

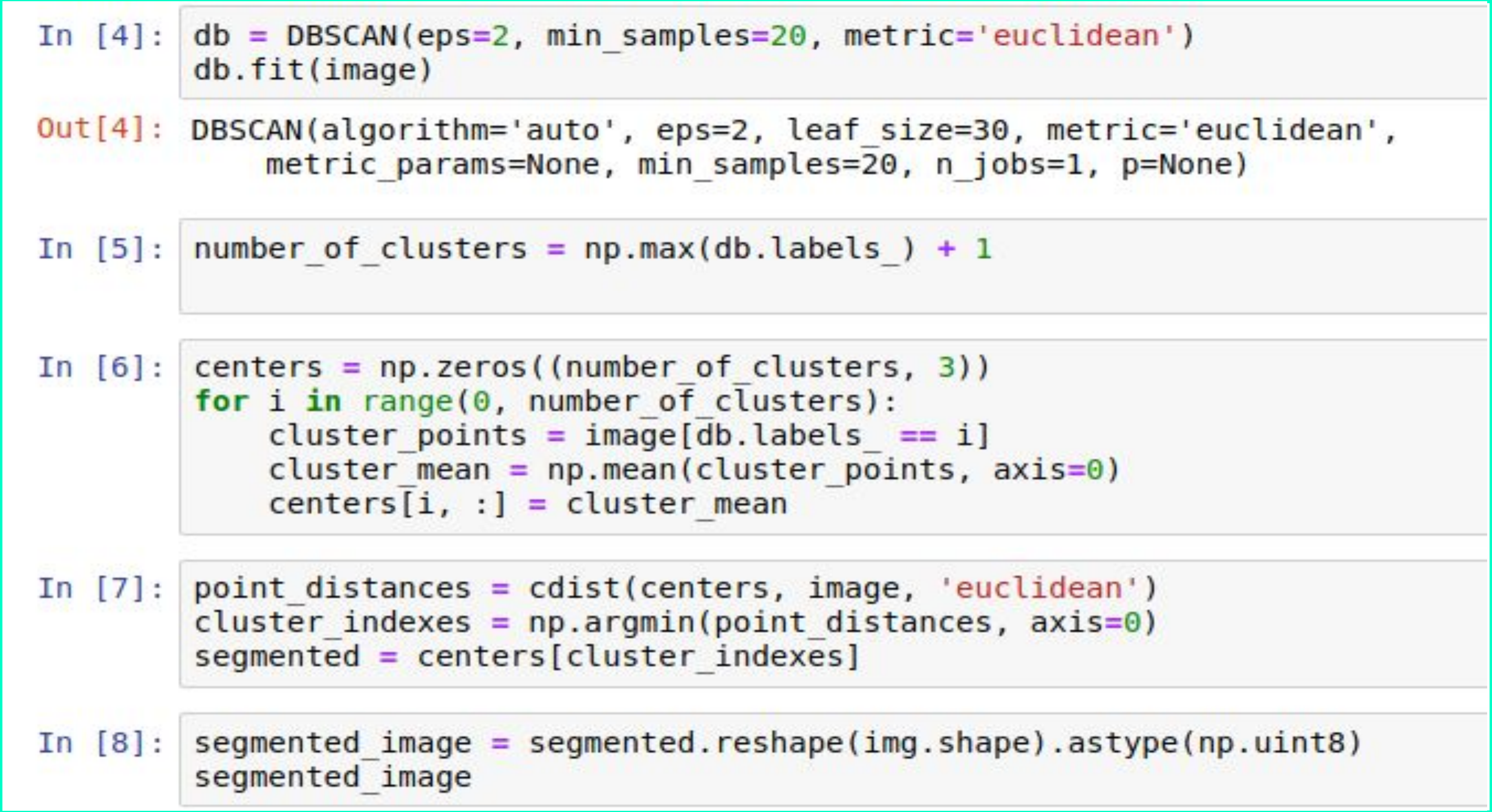
A noise point is any point that is not a core point or a border point

Pros

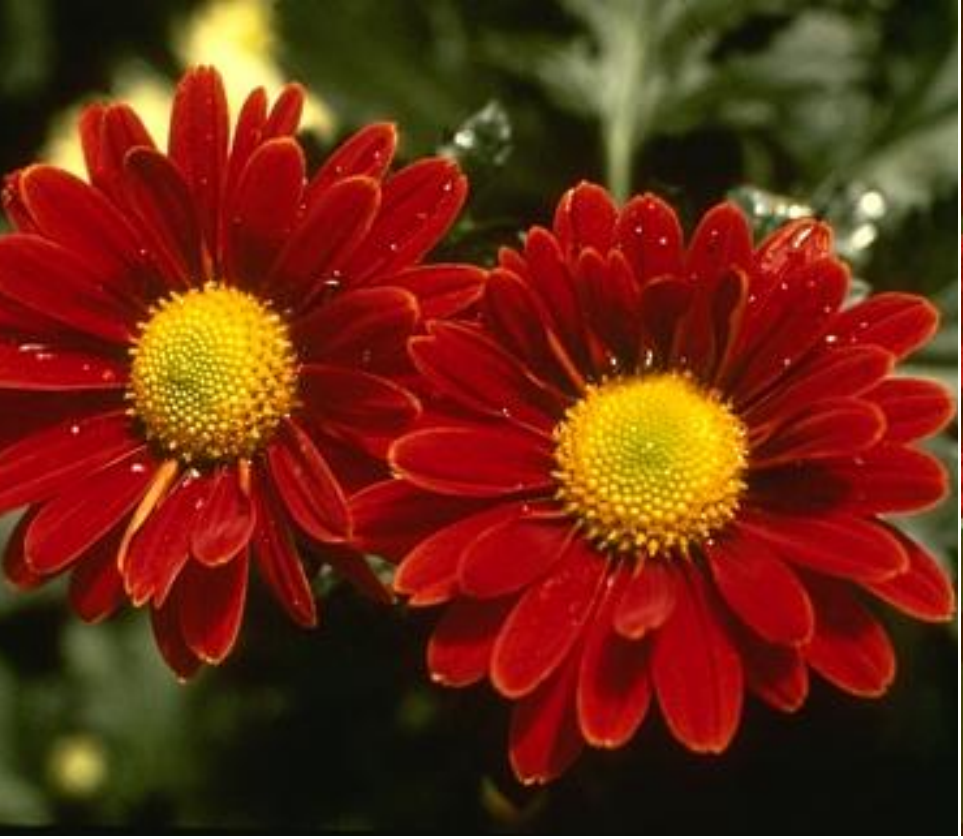
- Can discover arbitrary shaped clusters
- Robust towards outliers detection
- Does not require any prior K
- Beside K-means the second most used clustering algorithm

Cons

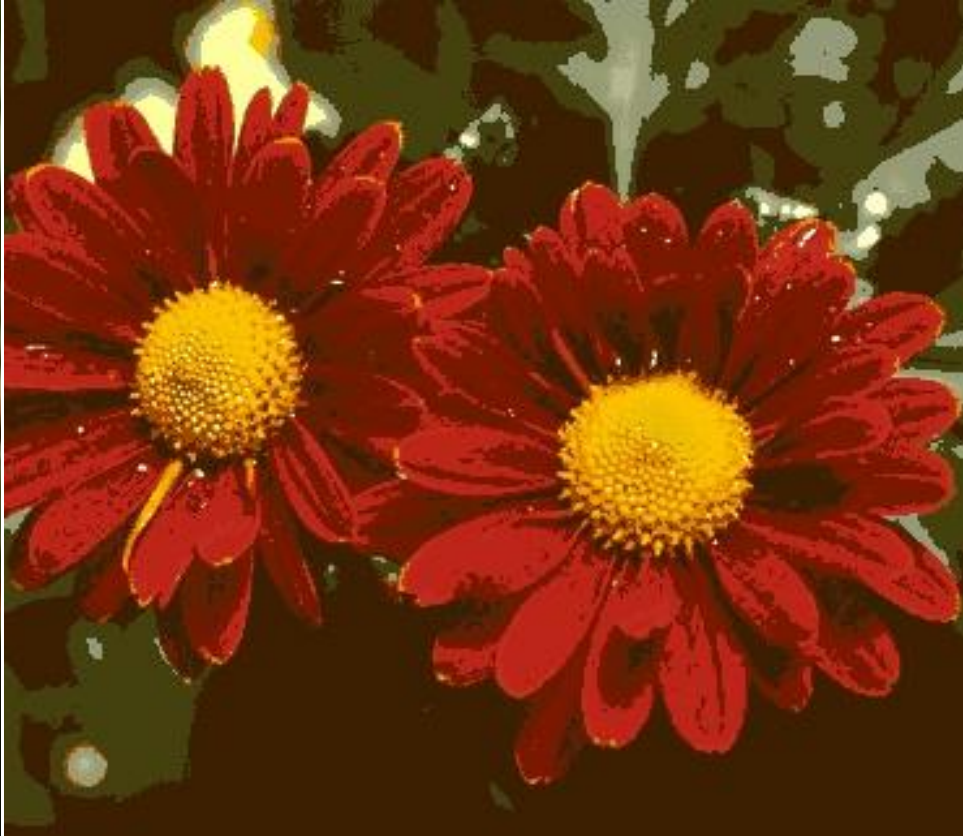
- Requires connected regions of sufficiently high density
- Data sets with varying densities are problematic
- Sensitive to clustering parameters EPS and MinPoints
- Does not work well in high-dimensional datasets



```
In [4]: db = DBSCAN(eps=2, min_samples=20, metric='euclidean')
        db.fit(image)

Out[4]: DBSCAN(algorithm='auto', eps=2, leaf_size=30, metric='euclidean',
            metric_params=None, min_samples=20, n_jobs=1, p=None)

In [5]: number_of_clusters = np.max(db.labels_) + 1

In [6]: centers = np.zeros((number_of_clusters, 3))
        for i in range(0, number_of_clusters):
            cluster_points = image[db.labels_ == i]
            cluster_mean = np.mean(cluster_points, axis=0)
            centers[i, :] = cluster_mean

In [7]: point_distances = cdist(centers, image, 'euclidean')
        cluster_indexes = np.argmin(point_distances, axis=0)
        segmented = centers[cluster_indexes]

In [8]: segmented_image = segmented.reshape(img.shape).astype(np.uint8)
        segmented_image
```



Before Dbscan



After Dbscan

MeanShift

MeanShift is a non parametric feature-space technique for locating the maxima of a density function, a so-called mode-seeking(centroid based) algorithm

It works by updating candidates for centroids to be the mean of the points within a given region

MeanShift Algorithm

Let a kernel function be given **K(Xi-X)**

This function determines the weight of nearby points for re-estimation of the mean

Typically a Gaussian kernel on the distance to the current estimate is used **K(Xi-X)** = **exp**(-c||**Xi-X**||2)

The weighted mean of the density in the window determined by K is

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

N(X) is the neighborhood of X, a set of points for which K(Xi) not equal to 0

The difference m(X)-X is called mean shift and meanshift algorithm sets m(X) to X and repeats the estimation until converges

Pros

- No assumptions on the shape or number of data clusters
- The procedure only has one parameter, bandwidth
- Output doesn’t depend on initializations

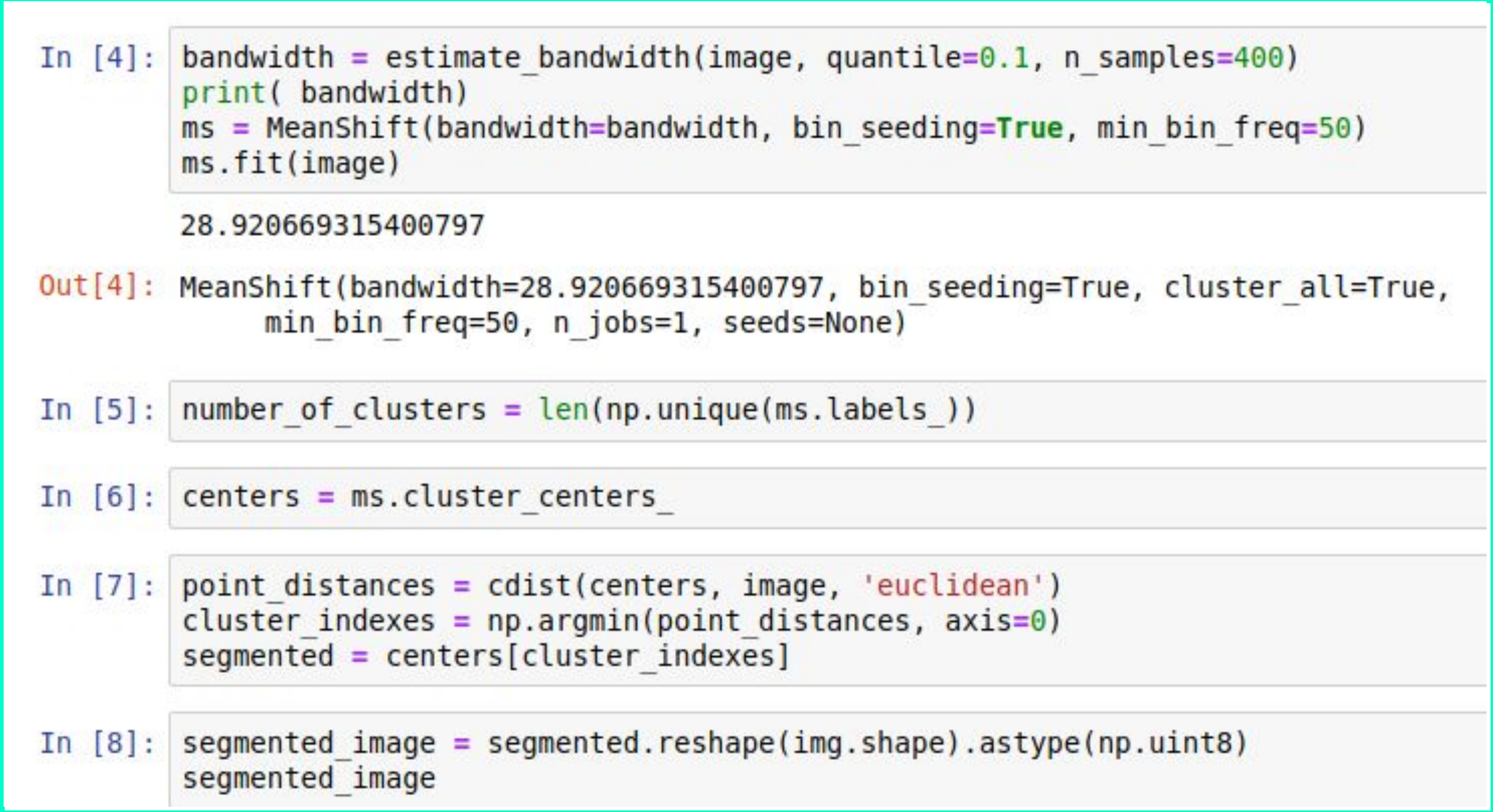
Cons

- Output does depend on bandwidth(too small=convergence is slow, too large=some clusters may be missed)
- Often slower than KMeans
- Computationally expensive for large feature spaces

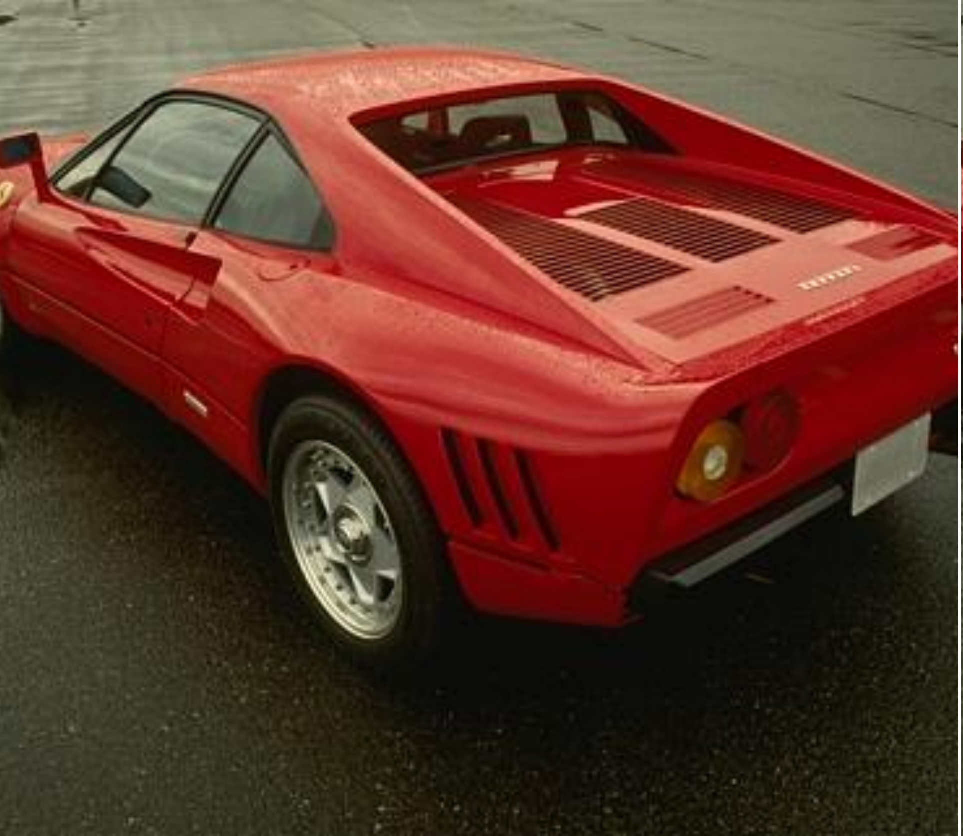
MeanShift Algorithm

Bandwidth parameter(which dictates the size of the region to search through) can be set manually, but can be estimated using the provide estimate_bandwidth function, which is called if the bandwidth is not set

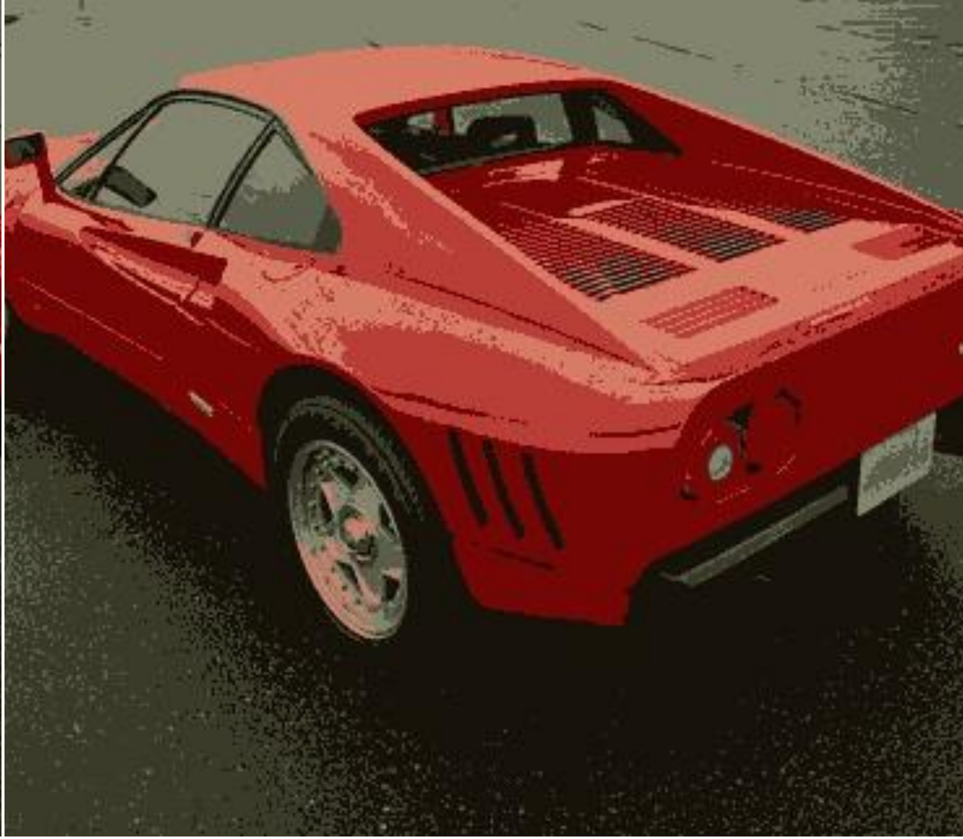
This is not highly scalable, as it requires multiple nearest neighbour searches during the execution of the algorithm



```
In [4]: bandwidth = estimate_bandwidth(image, quantile=0.1, n_samples=400)
        print( bandwidth)
        ms = MeanShift(bandwidth=bandwidth, bin_seeding=True, min_bin_freq=50)
        ms.fit(image)

        28.920669315400797

Out[4]: MeanShift(bandwidth=28.920669315400797, bin_seeding=True, cluster_all=True,
             min_bin_freq=50, n_jobs=1, seeds=None)

In [5]: number_of_clusters = len(np.unique(ms.labels_))

In [6]: centers = ms.cluster_centers_

In [7]: point_distances = cdist(centers, image, 'euclidean')
        cluster_indexes = np.argmin(point_distances, axis=0)
        segmented = centers[cluster_indexes]

In [8]: segmented_image = segmented.reshape(img.shape).astype(np.uint8)
        segmented_image
```



Before MeanShift



After MeanShift

Gaussian Mixture Model

A probabilistic approach to clustering in which we describe each cluster by its centroid(mean), covariance, and the size of the cluster(Weight)

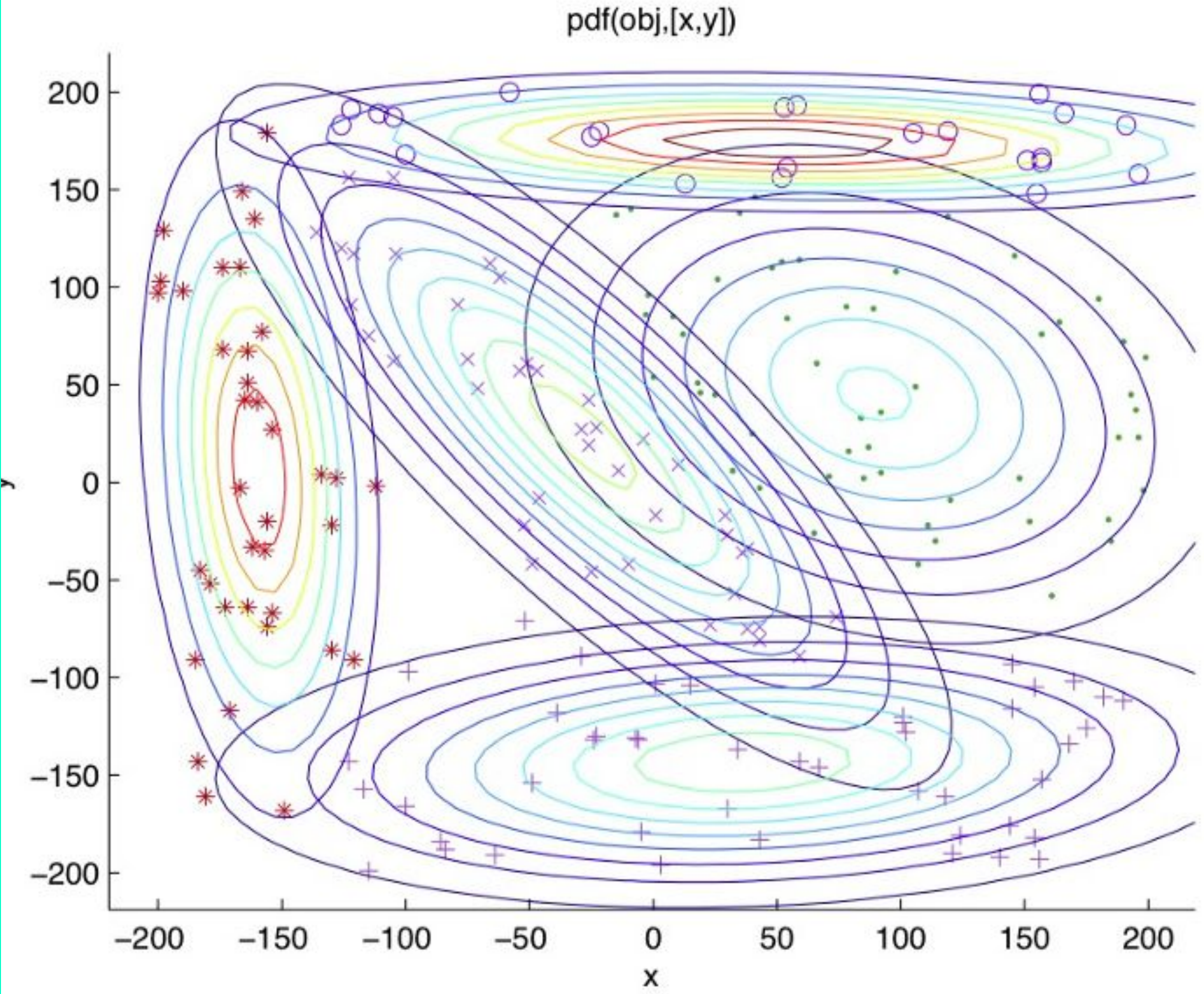
Gaussian Mixture Model

Rather than identifying clusters by “nearest” centroids, we fit a set of k gaussians to the data

We estimate gaussian distribution parameters such as mean and variance for each each cluster and weight of a cluster

After learning the parameters for each point we can calculate the probabilities of it belonging to each of the clusters

After getting component distribution, depending upon variance and mean of the particular cluster we get probability of any data point x belongs to every cluster

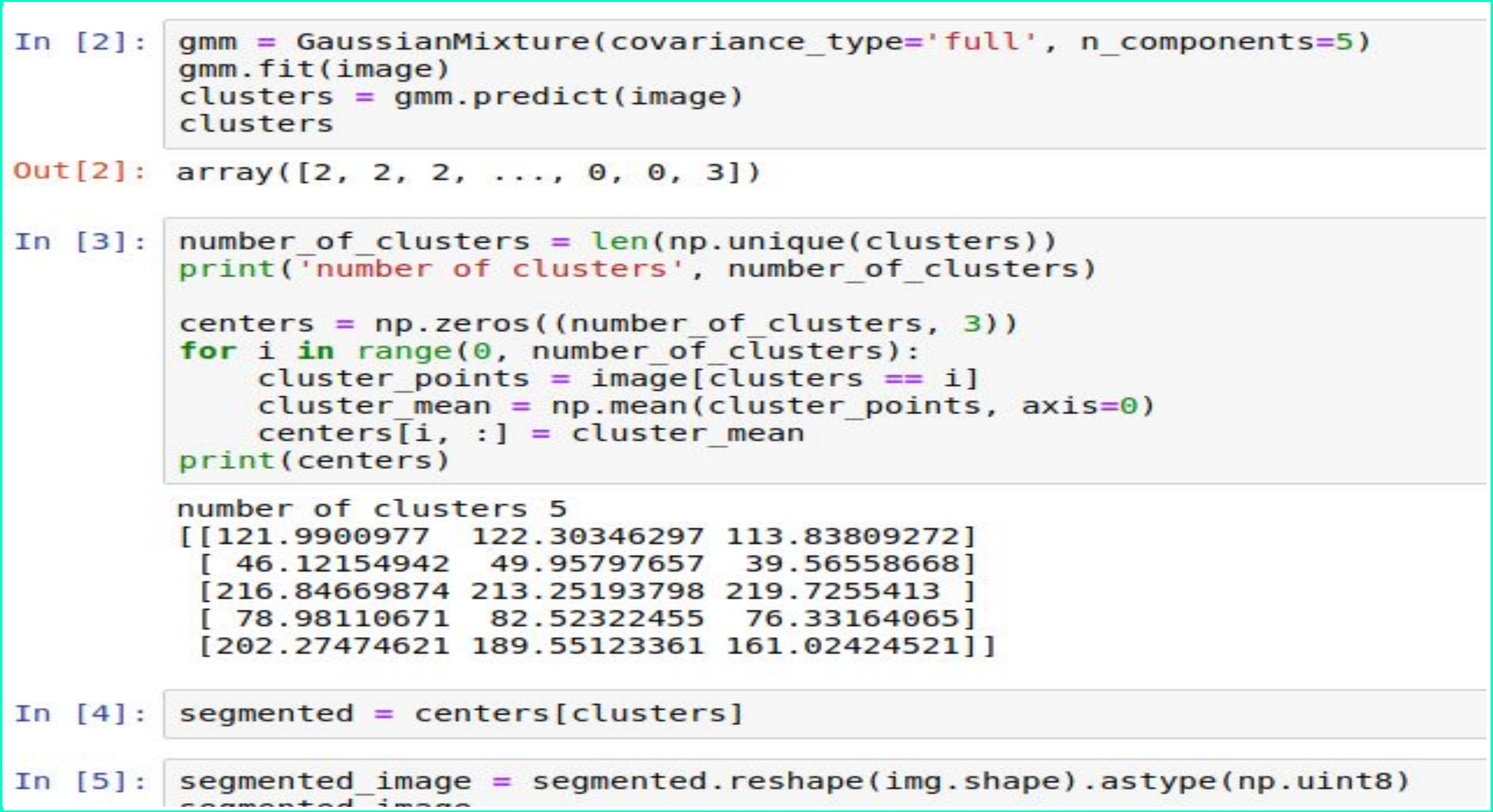


Gmm

KMeans is a special case of GMM, such that probability of a one point to belong to a certain cluster is 1, and all other probabilities are 0, and the variance is 1, which is a reason why KMeans produces only spherical clusters

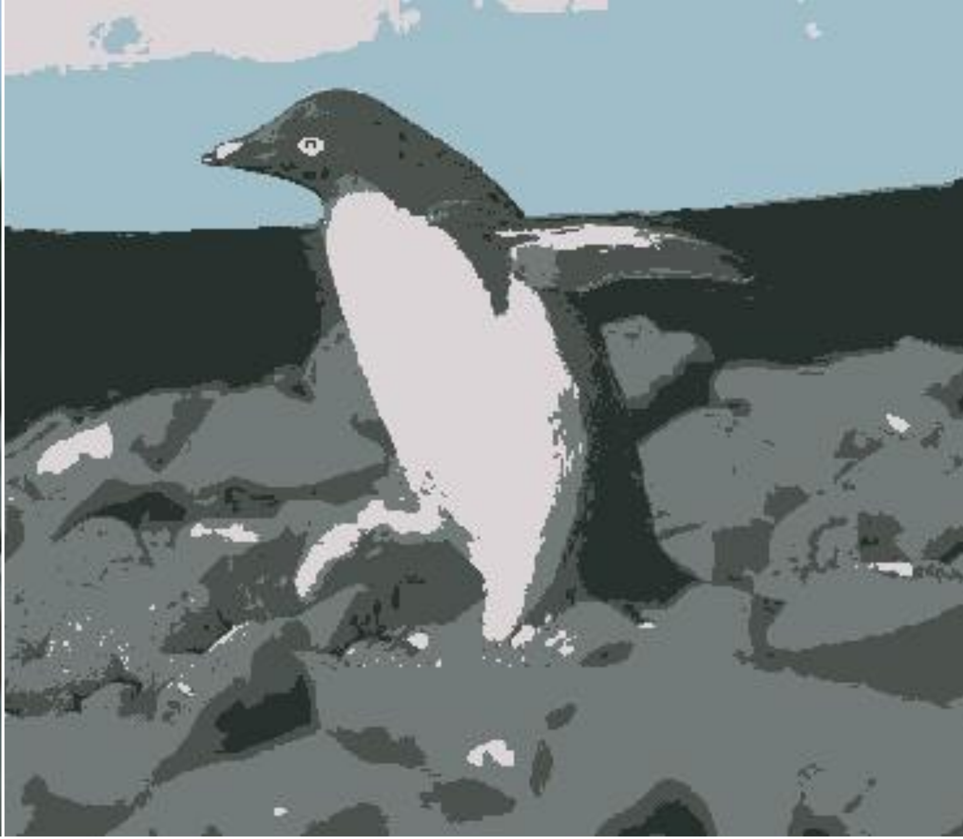
GMM produces non-convex clusters, which can be controlled with the variance of the distribution

The algorithms for optimizing the loss function for GMM are not so trivial, since it is not a convex function. The most popular algorithm is the Expectation Maximization algorithm



```
In [2]: gmm = GaussianMixture(covariance_type='full', n_components=5)
        gmm.fit(image)
        clusters = gmm.predict(image)
        clusters

Out[2]: array([2, 2, 2, ..., 0, 0, 3])

In [3]: number_of_clusters = len(np.unique(clusters))
        print('number of clusters', number_of_clusters)

        centers = np.zeros((number_of_clusters, 3))
        for i in range(0, number_of_clusters):
            cluster_points = image[clusters == i]
            cluster_mean = np.mean(cluster_points, axis=0)
            centers[i, :] = cluster_mean
        print(centers)

        number of clusters 5
        [[121.9900977  122.30346297 113.83809272]
         [ 46.12154942  49.95797657  39.56558668]
         [216.84669874 213.25193798 219.7255413 ]
         [ 78.98110671  82.52322455  76.33164065]
         [202.27474621 189.55123361 161.02424521]]

In [4]: segmented = centers[clusters]

In [5]: segmented_image = segmented.reshape(img.shape).astype(np.uint8)
```

Before Gmm



After Gmm

Agglomearative Clustering

Agglomerative Clustering is a Hierarchical Clustering algorithm

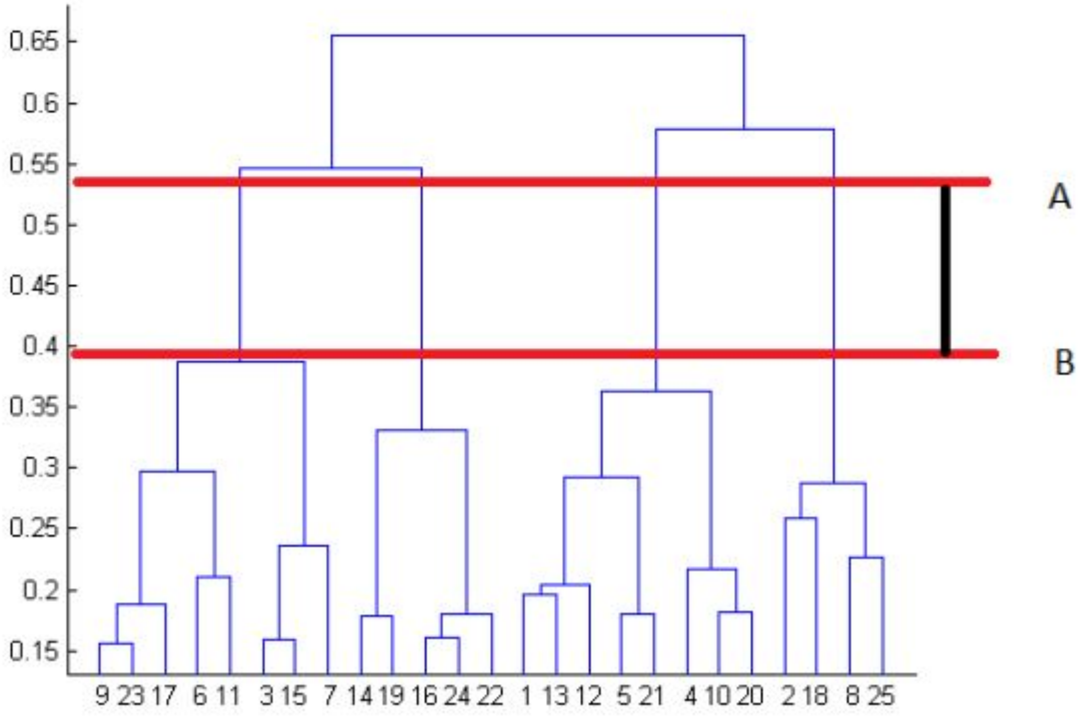
This algorithm starts with all the data points assigned to a cluster of their own. Then two nearest clusters are merged into the same cluster. In the end, this algorithm terminates when there is only a single cluster left

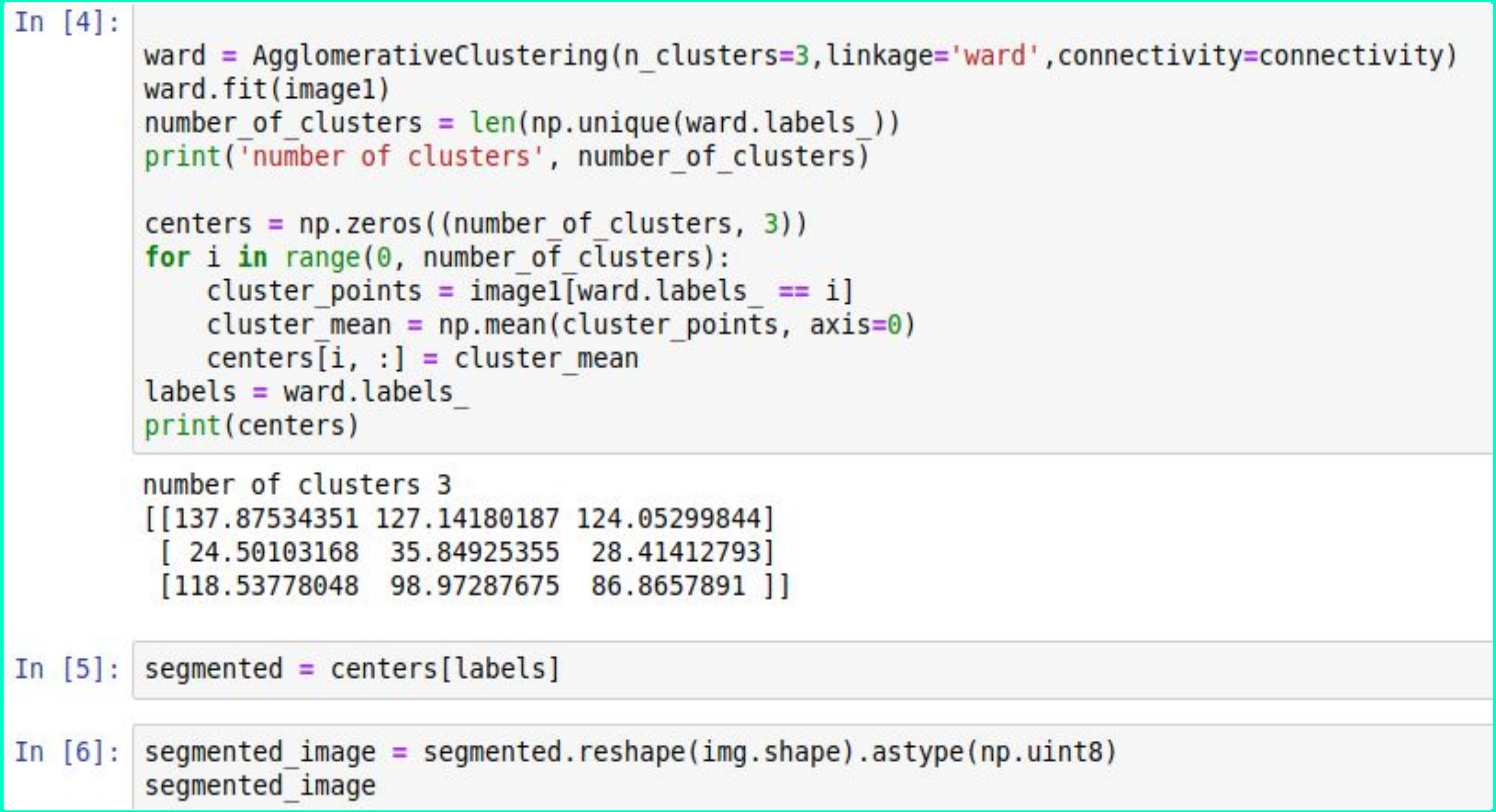
The decision of merging two clusters is taken on the basis of closeness of these clusters. There are multiple metrics for deciding the closeness of two clusters:

- Euclidean Distance:: $||a-b||2 = \surd(\Sigma(ai-bi))$
- Squared Euclidean distance:$||a-b||22 = \Sigma((ai-bi)2)$
- Maximum distance:$maxi|ai-bi|$
- Mahalanobis distance:$\surd((a-b)T\ S-1\ (-b))$\{where, s : covariance matrix\}

Agglomerative hierarchical clustering techniques perform clustering on a local level and as such there is no global objective function like in the K-Means algorithm

Dendrogram


0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
A
B
9 23 17 6 11 3 15 7 14 19 16 24 22 1 13 12 5 21 4 10 20 2 18 8 25

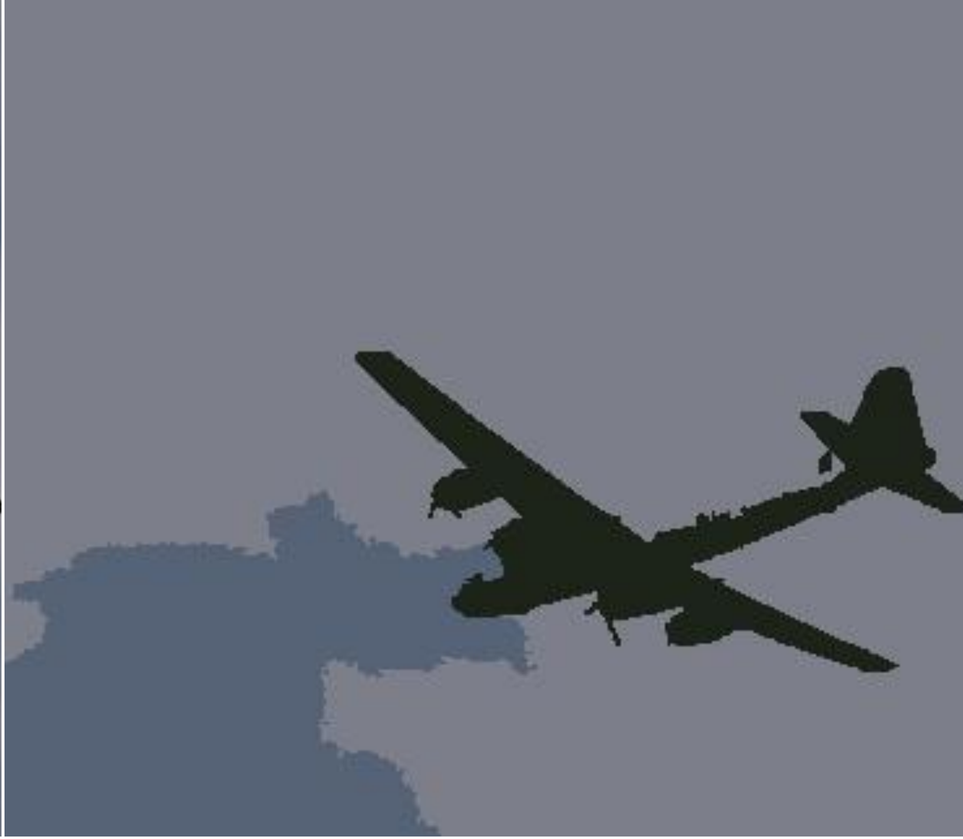


```
In [4]:
ward = AgglomerativeClustering(n_clusters=3,linkage='ward',connectivity=connectivity)
ward.fit(image1)
number_of_clusters = len(np.unique(ward.labels_))
print('number of clusters', number_of_clusters)

centers = np.zeros((number_of_clusters, 3))
for i in range(0, number_of_clusters):
    cluster_points = image1[ward.labels_ == i]
    cluster_mean = np.mean(cluster_points, axis=0)
    centers[i, :] = cluster_mean
labels = ward.labels_
print(centers)
```

```
number of clusters 3
[[137.87534351 127.14180187 124.05299844]
 [ 24.50103168  35.84925355  28.41412793]
 [118.53778048  98.97287675  86.8657891 ]]
```

```
In [5]: segmented = centers[labels]
```

```
In [6]: segmented_image = segmented.reshape(img.shape).astype(np.uint8)
        segmented_image
```

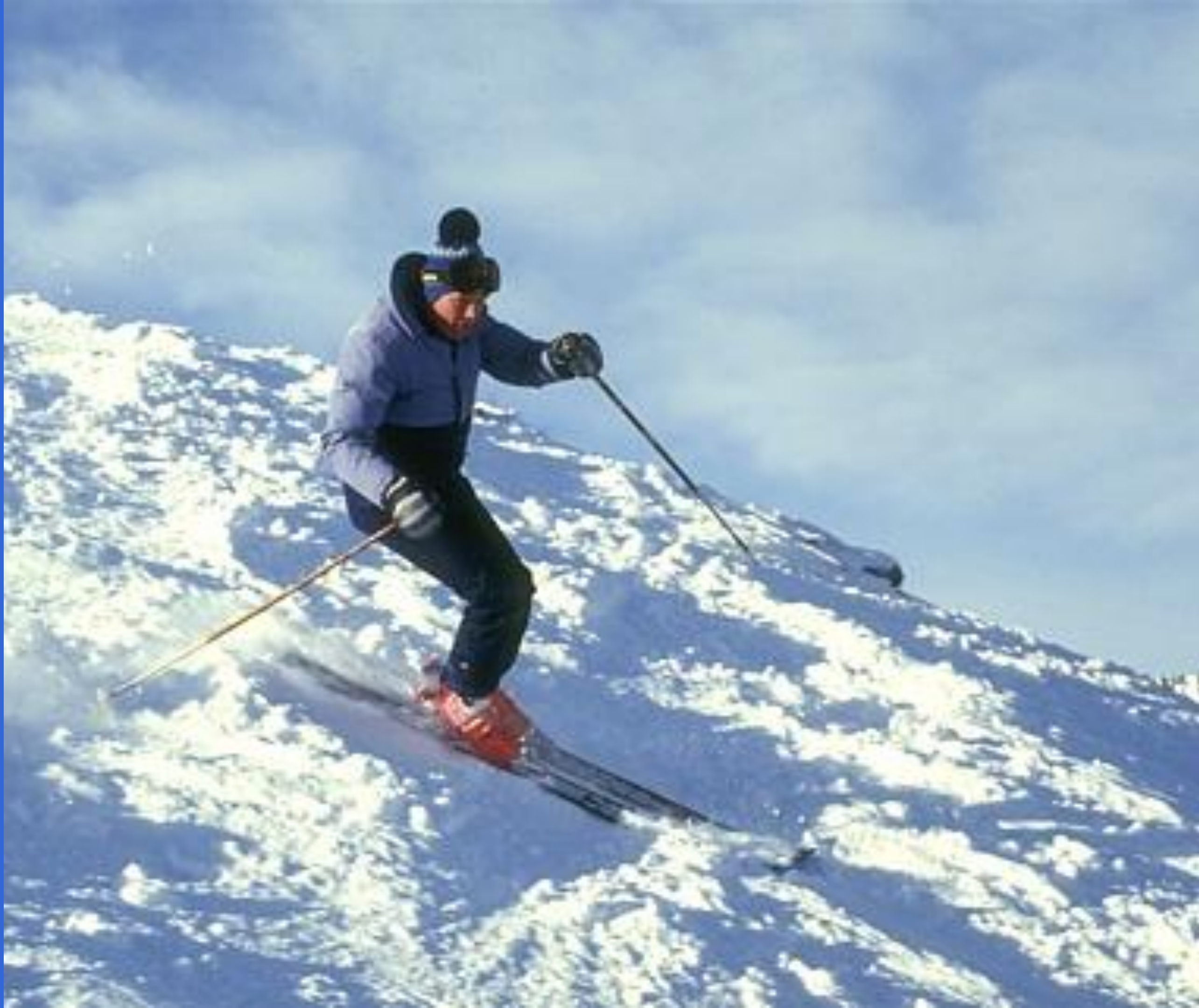


Before Agglomerative

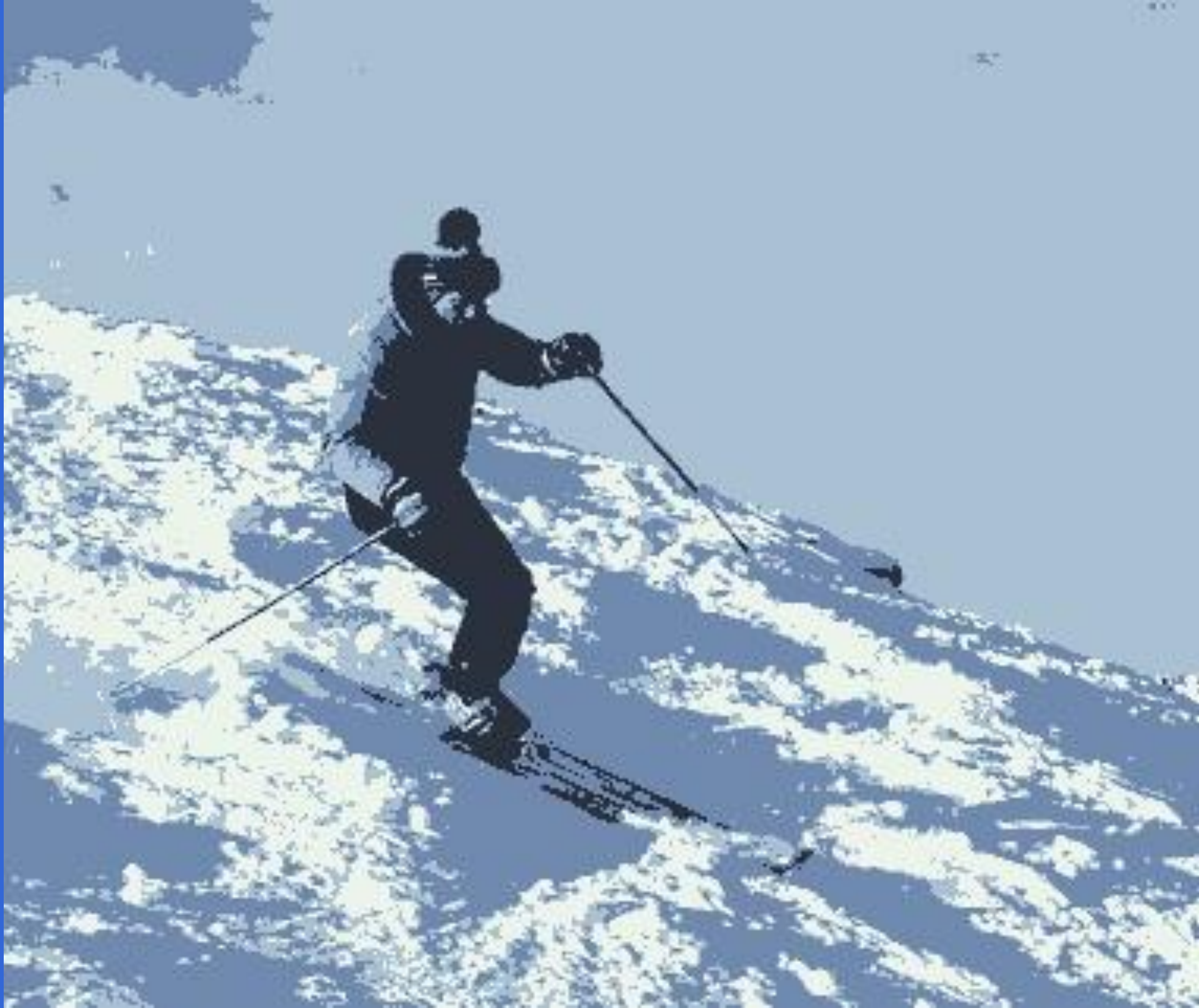
After Agglomerative

Now we will see how all these algorithms does image segmentation on an image

Image Before Clustering



Kmeans



MeanShift

DBSCAN

Gaussian Mixture Model

Agglomerative Clustering