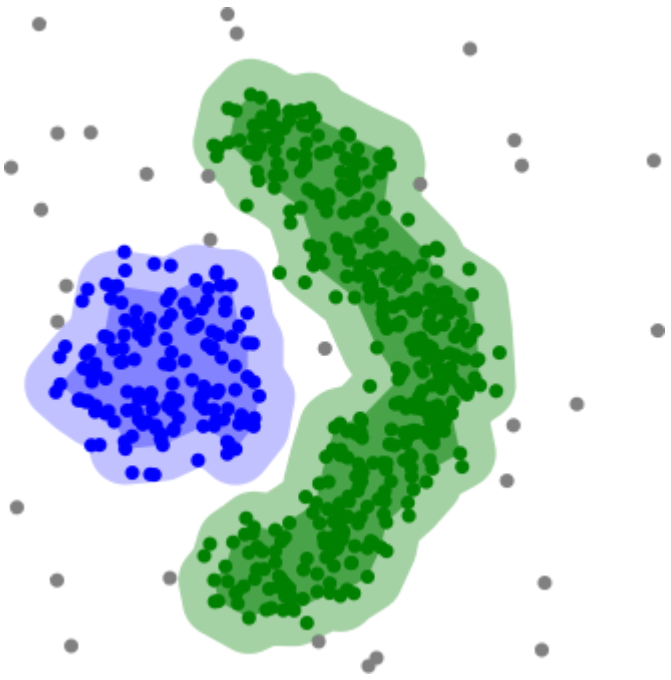
Clustering Activity Project 1

Noe Lomidze

13.10.2024

DBSCAN can find non-linearly separable clusters. This dataset cannot be adequately clustered with k-means, k medoids or Gaussian Mixture EM clustering.



Disadvantages

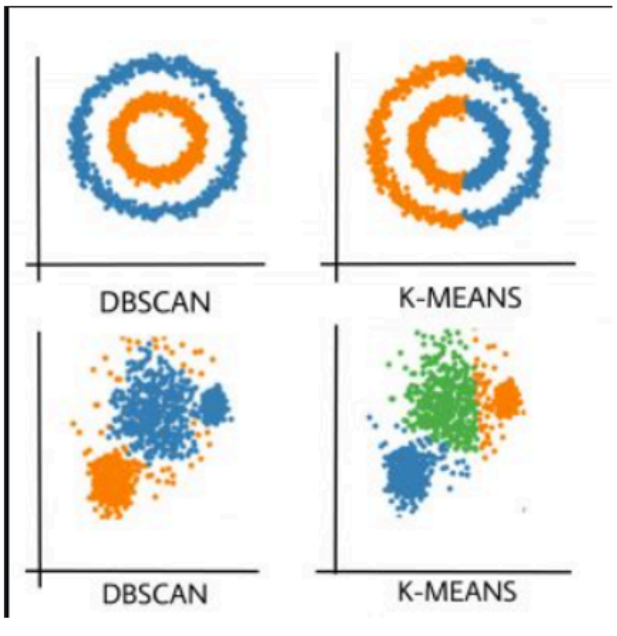
1. DBSCAN is not entirely **deterministic**: border points that are reachable from more than one cluster **can be part of either cluster**, depending on the order the data are processed. For most data sets and domains, this situation does not arise often and has little impact on the clustering result: both on core points and noise points, DBSCAN is deterministic. DBSCAN is a variation that **treats border points as noise**, and this way achieves a fully deterministic result as well as a more consistent statistical interpretation of density-connected components.
2. The quality of DBSCAN depends on the **distance measure** used in the function regionQuery(P,ε). The most common distance metric used is **Euclidean distance**. Especially for high-dimensional data, this metric can be rendered almost useless due to the so-called Curse of dimensionality, making it difficult to find an appropriate value for ε. This effect, however, is also present in any other algorithm based on Euclidean distance.
3. **DBSCAN cannot cluster data sets well with large differences in densities**, since the **minPts**-ε combination cannot then be chosen appropriately for all clusters.
4. If the data and scale are not well understood, choosing a meaningful distance threshold ε can be difficult.

```
_class_ sklearn.cluster.DBSCAN(_eps=0.5_, _*_, _min_samples=5_, _metric='euclidean'_, _metric_params=None_,
_algorithm='auto'_, _leaf_size=30_, _p=None_, _n_jobs=None_)
```

which can occur when the `eps` parameter is large and `min_samples` is low, while the original DBSCAN only uses linear memory. For further details, see the Notes below.

DBSCAN Clustering

Clusters are dense regions in the data space, separated by regions of the lower density of points. The *DBSCAN algorithm* is based on this intuitive notion of “clusters” and “noise”. The key idea is that for each point of a cluster, the neighborhood of a given radius has to contain at least a minimum number of points.



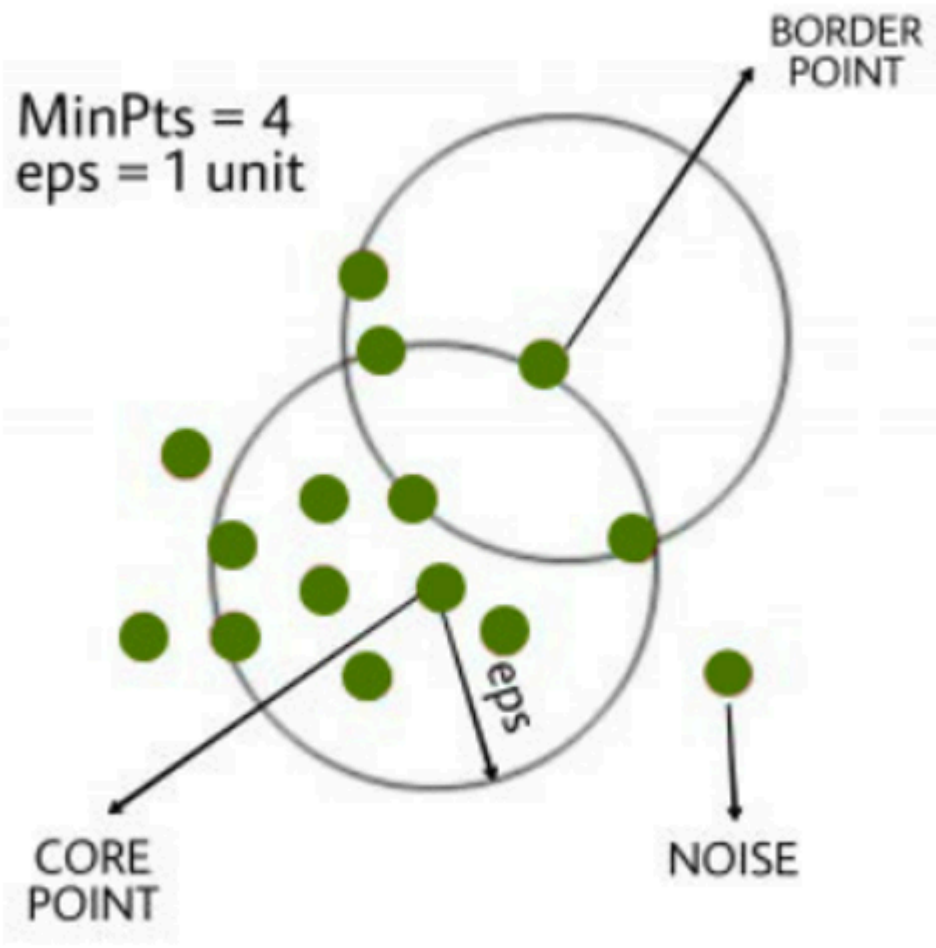
Partitioning methods (K-means, PAM clustering) and hierarchical clustering work for finding spherical-shaped clusters or convex clusters. In other words, they are suitable only for compact and well-separated clusters. Moreover, they are also severely affected by the presence of noise and outliers in the data.

Real-life data may contain irregularities, like:

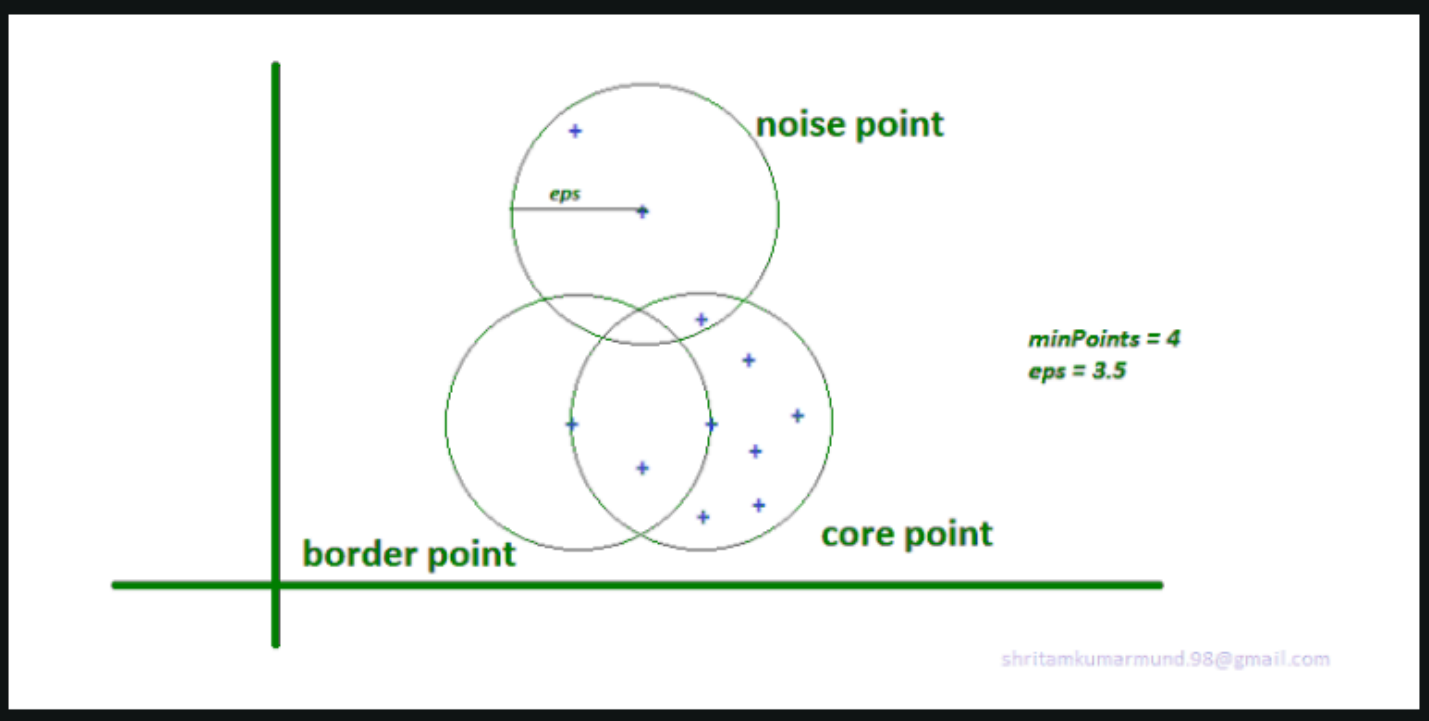
- Clusters can be of arbitrary shape such as those shown in the figure below.
- Data may contain noise.

Required parameters

- eps: It defines the neighborhood around a data point i.e. if the distance between two points is lower or equal to ‘eps’ then they are considered neighbors. If the eps value is chosen too small then a large part of the data will be considered as an outlier. If it is chosen very large then the clusters will merge and the majority of the data points will be in the same clusters. One way to find the eps value is based on the k-distance graph.
- MinPts: Minimum number of neighbors (data points) within eps radius. The larger the dataset, the larger value of MinPts must be chosen. As a general rule, the minimum MinPts can be derived from the number of dimensions D in the dataset as, MinPts >= D+1. The minimum value of MinPts must be chosen at least 3.



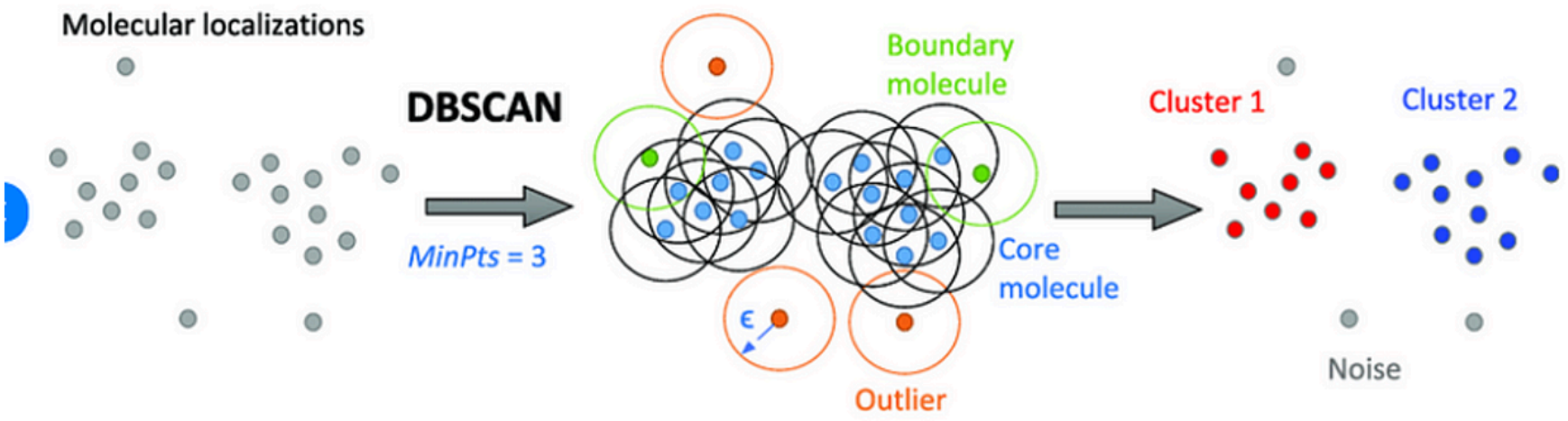
DBSCAN Architecture:



Eps : Radius of circle **minPts** : It is the minimum no. of points that must exist in the vicinity of eps.

1. Find all the neighbor points within eps and identify the core points or visited with more than MinPts neighbors.
2. For each core point if it is not already assigned to a cluster, create a new cluster.
3. Find recursively all its density-connected points and assign them to the same cluster as the core point.
   A point *a* and *b* are said to be density connected if there exists a point *c* which has a sufficient number of points in its neighbors and both points *a* and *b* are within the *eps distance*. This is a chaining process. So, if *b* is a neighbor of *c*, *c* is a neighbor of *d*, and *d* is a neighbor of *e*, which in turn is neighbor of *a* implying that *b* is a neighbor of *a*.
4. Iterate through the remaining unvisited points in the dataset. Those points that do not belong to any cluster are noise.

Pseudocode For DBSCAN Clustering Algorithm



K-medoids

Algorithm:

Given the value of k and unlabelled data:

1. Choose k number of random points from the data and assign these k points to k number of clusters. These are the initial medoids.
2. For all the remaining data points, calculate the distance from each medoid and assign it to the cluster with the nearest medoid.
3. Calculate the total cost (Sum of all the distances from all the data points to the medoids)
4. Select a random point as the new medoid and swap it with the previous medoid. Repeat 2 and 3 steps.
5. If the total cost of the new medoid is less than that of the previous medoid, make the new medoid permanent and repeat step 4.
6. If the total cost of the new medoid is greater than the cost of the previous medoid, undo the swap and repeat step 4.
7. The Repetitions have to continue until no change is encountered with new medoids to classify data points.

k-medoids is much more expensive

That's the main drawback. Usually, PAM takes much longer to run than k-means. As it involves computing all pairwise distances, it is `O(n^2*k*i)`; whereas k-means runs in `O(n*k*i)` where usually, k times the number of iterations is `k*i << n`

> Related to k-means

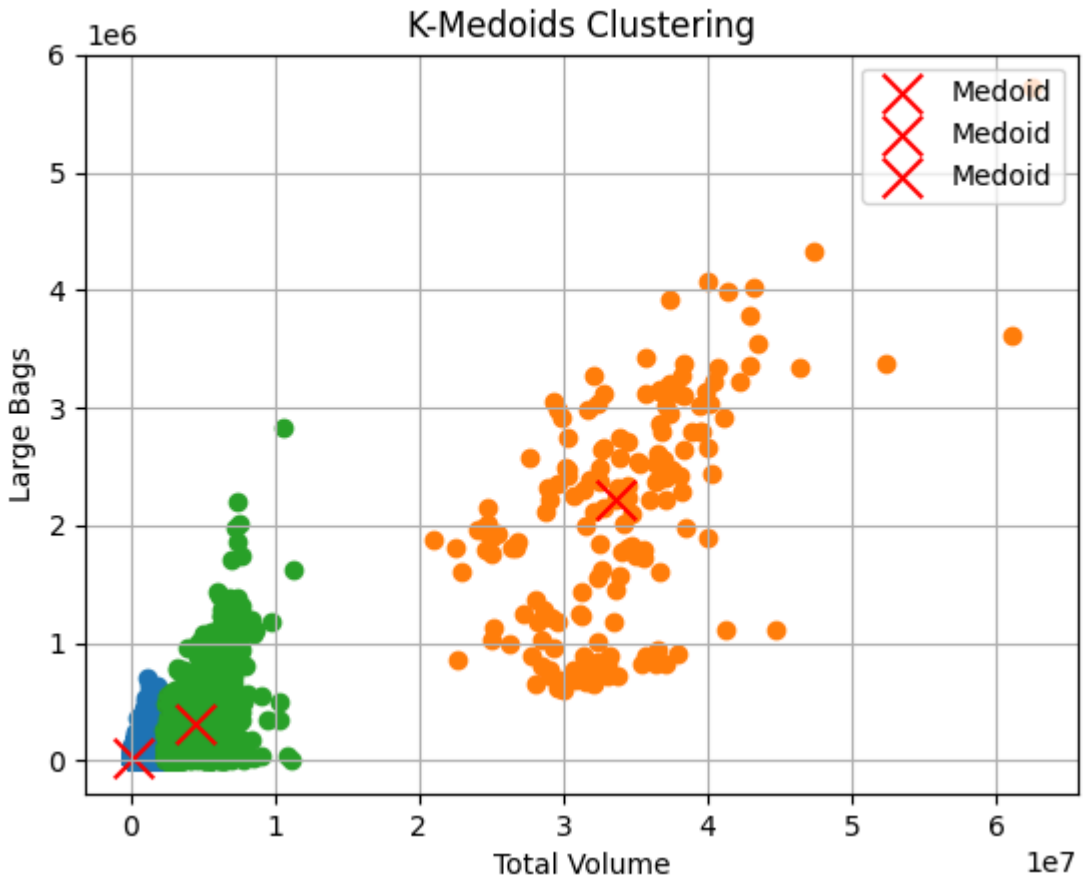
So here I start working at the project

At first let's test K-medoids algorithm with some random data

I used some random avocado data from Kaggle

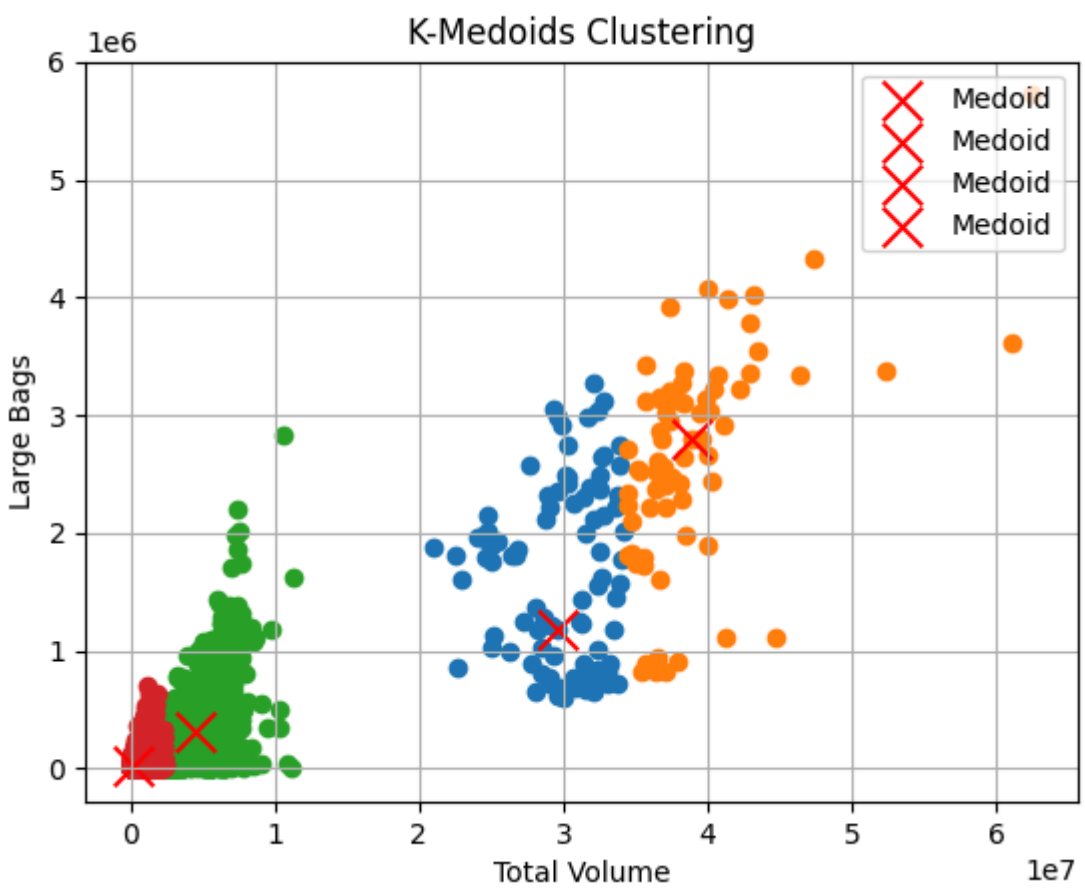
```
df = pd.read_csv('avocado.csv')

data = df[['Total Volume', 'Large Bags']].values
```

> And cluster that data with respect to two components of Avocado

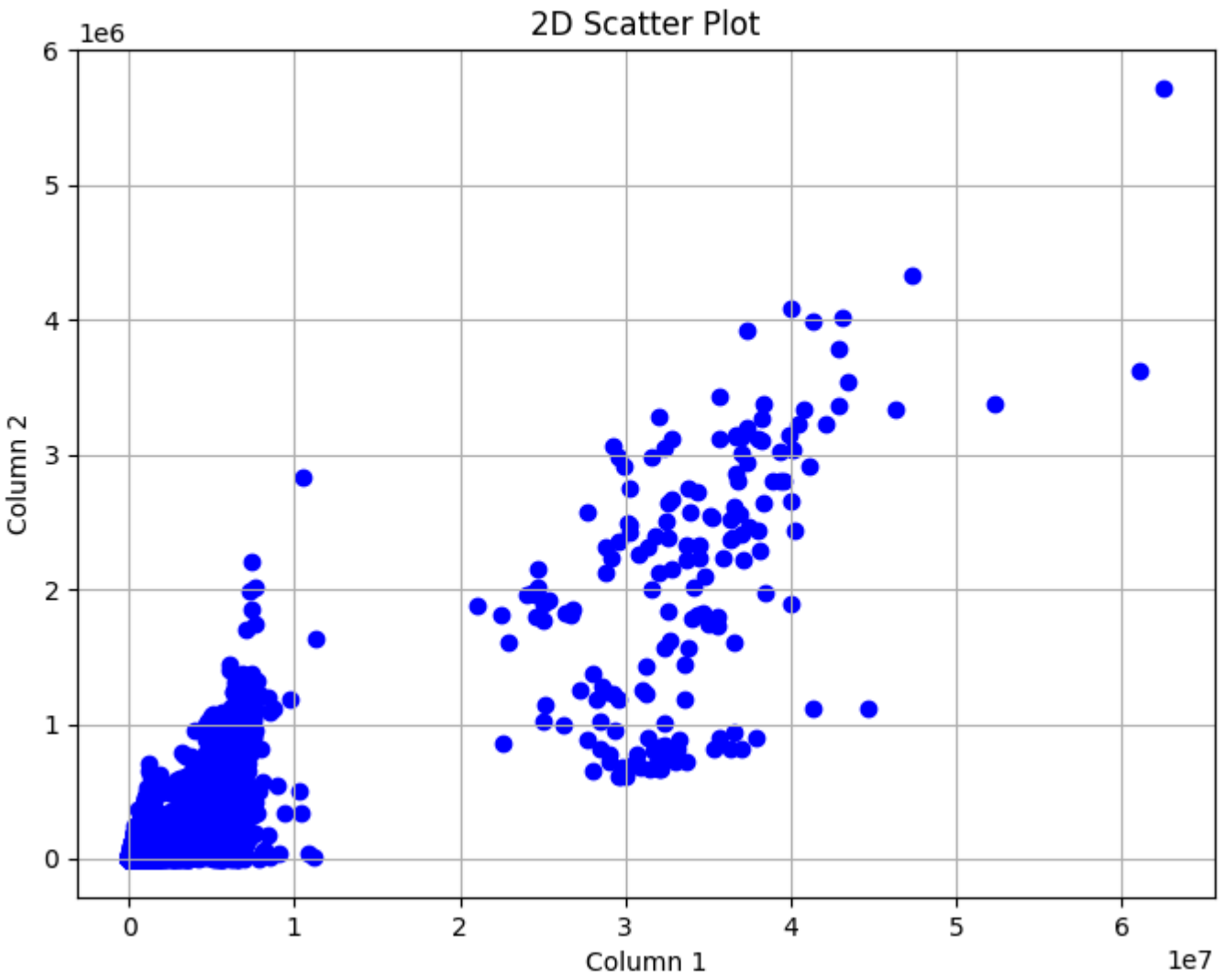


As you can see the result is not really impressive, although it took algorithm forever to run...

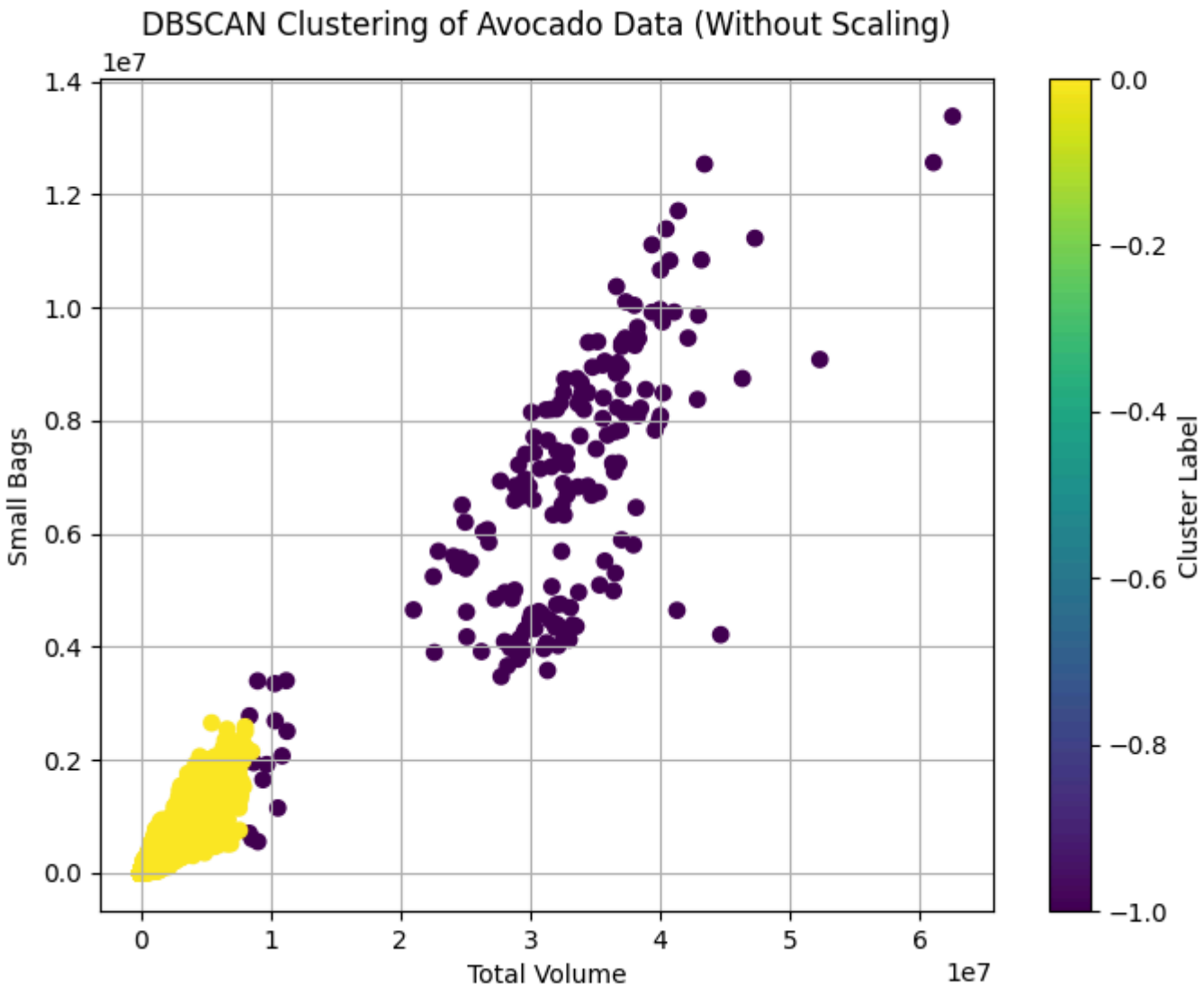
Result when clusters = 4



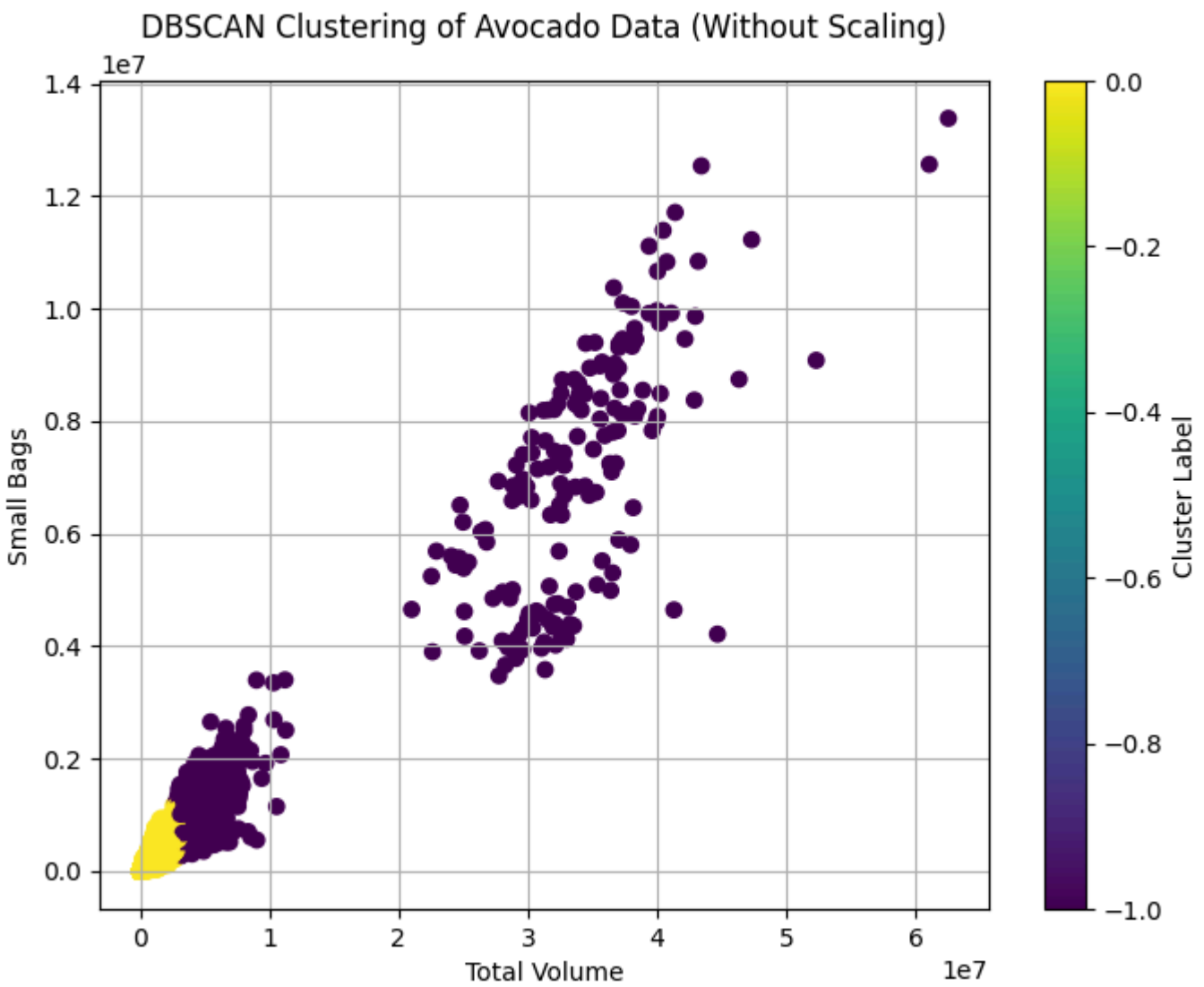
This is just the data plotted



Now let's quickly check how DBSCAN algorithm works on that data:



eps = 1_000_000, min_samples = 100



eps = 1_000_000, min_samples = 1000
I'm sure you can see the difference between min_samples 100 and 1000

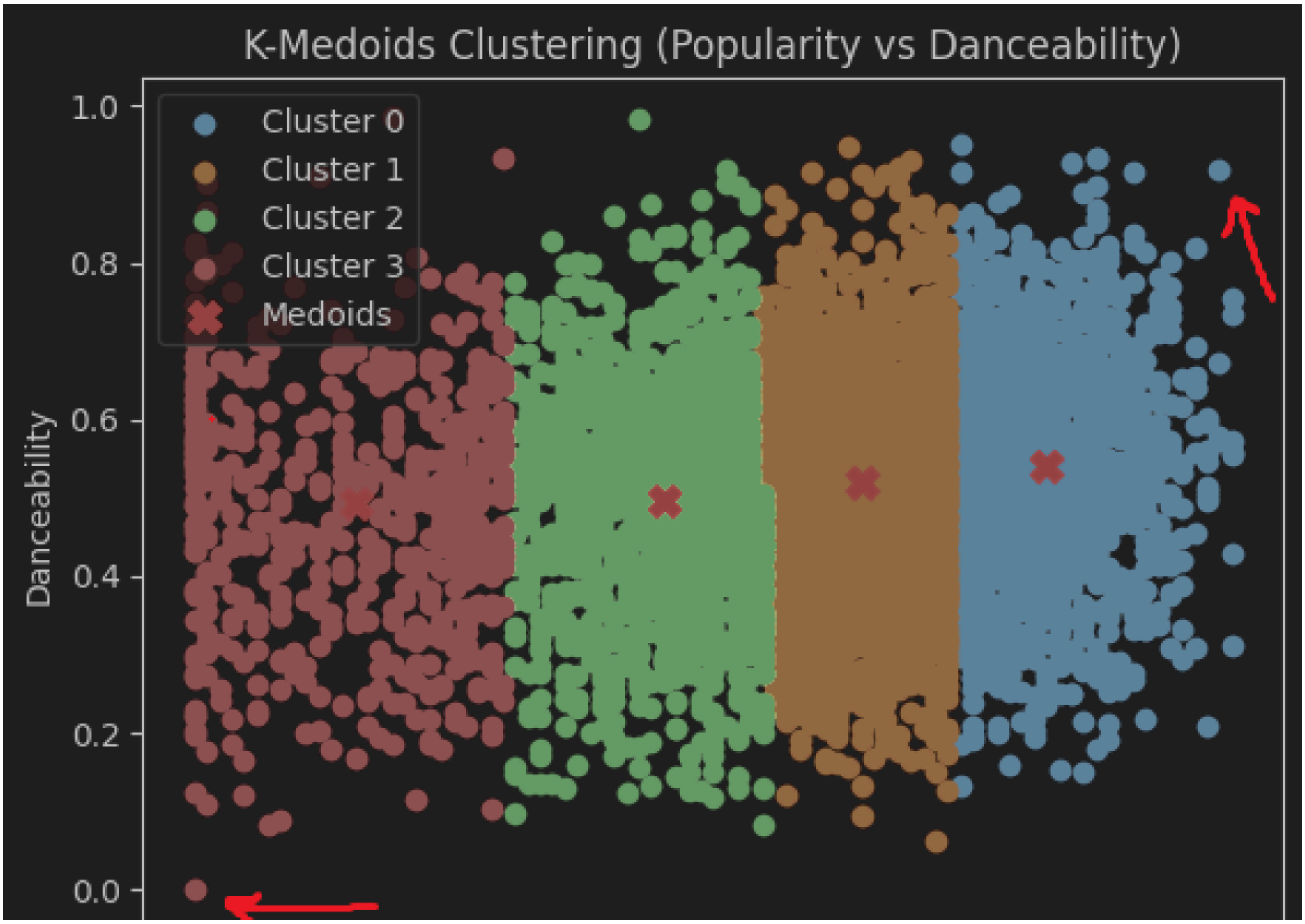
Here I start working on my main data:

My dataset at first was about Rock songs with columns:

- name
- artist
- release date
- popularity
- danceability

…

I took two of those, popularity and danceability and cluster songs respect to those two criteria



> The result was like that

If you are curious about those to points, the most popular song which is also high in danceability is and vice versa are:

```
Farthest Song:
Name: Jingle Bell Rock, Artist: Bobby Helms, Distance: 84.00338395564788

Nearest Song:
Name: You Wreck Me - Commentary, Artist: Various Artists, Distance: 0.0
```

> Jingle Bell Rock is definitely a good song

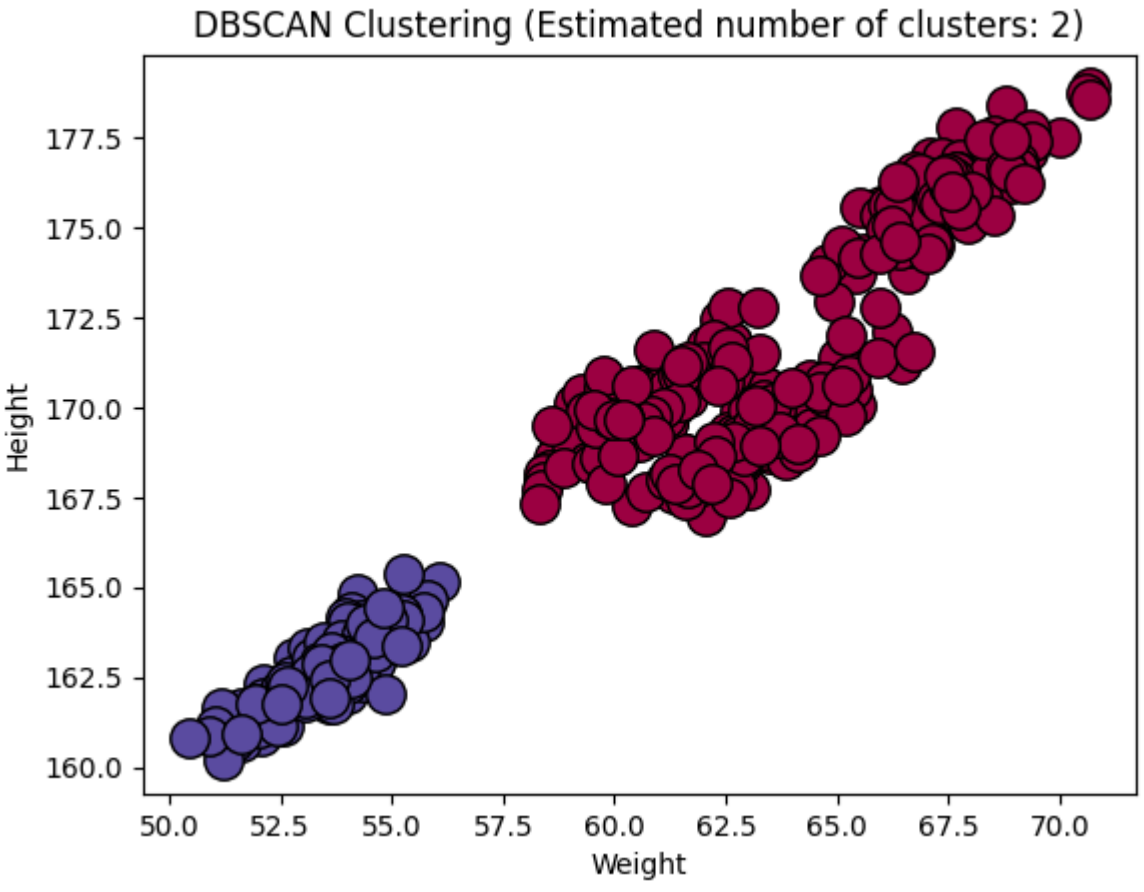
Runtime of K-medoids on that data was a bit expensive, in my case 10s 293ms

Now let's move on the DBSCAN algorithm

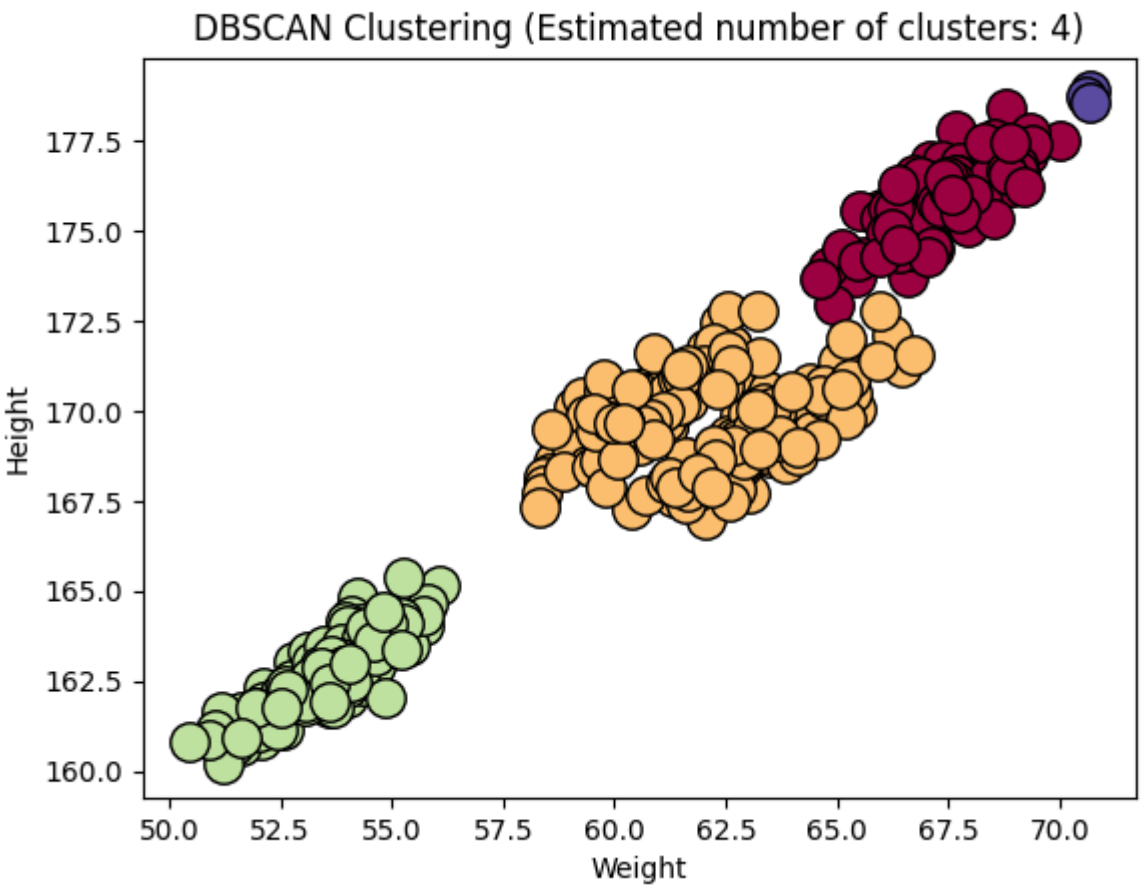
DBSCAN algorithm is really sensitive when it comes to changing its parameters:
eps and min_samples.

At first let's take a look at examples of DBSCAN on different data (randomly generated by me)

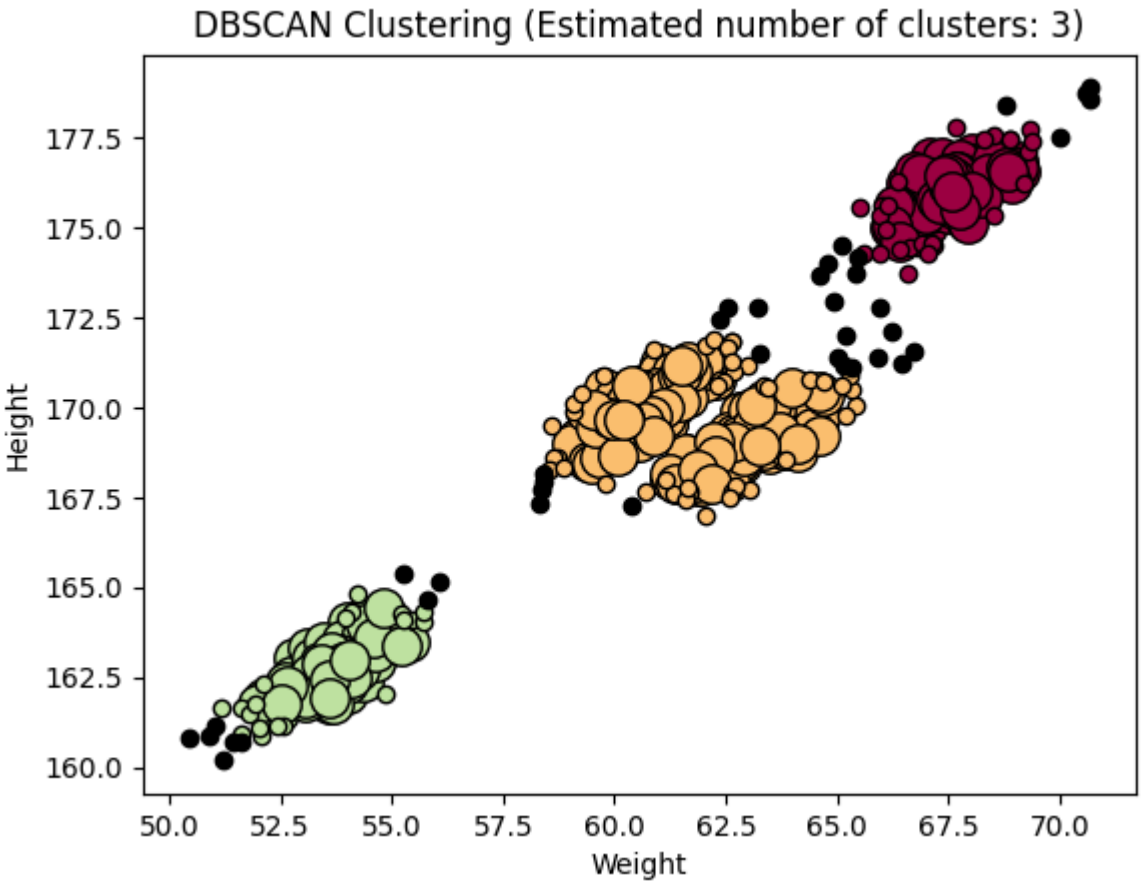
eps = 1.5, min_samples = 2



eps = 0.9, min_samples = 2

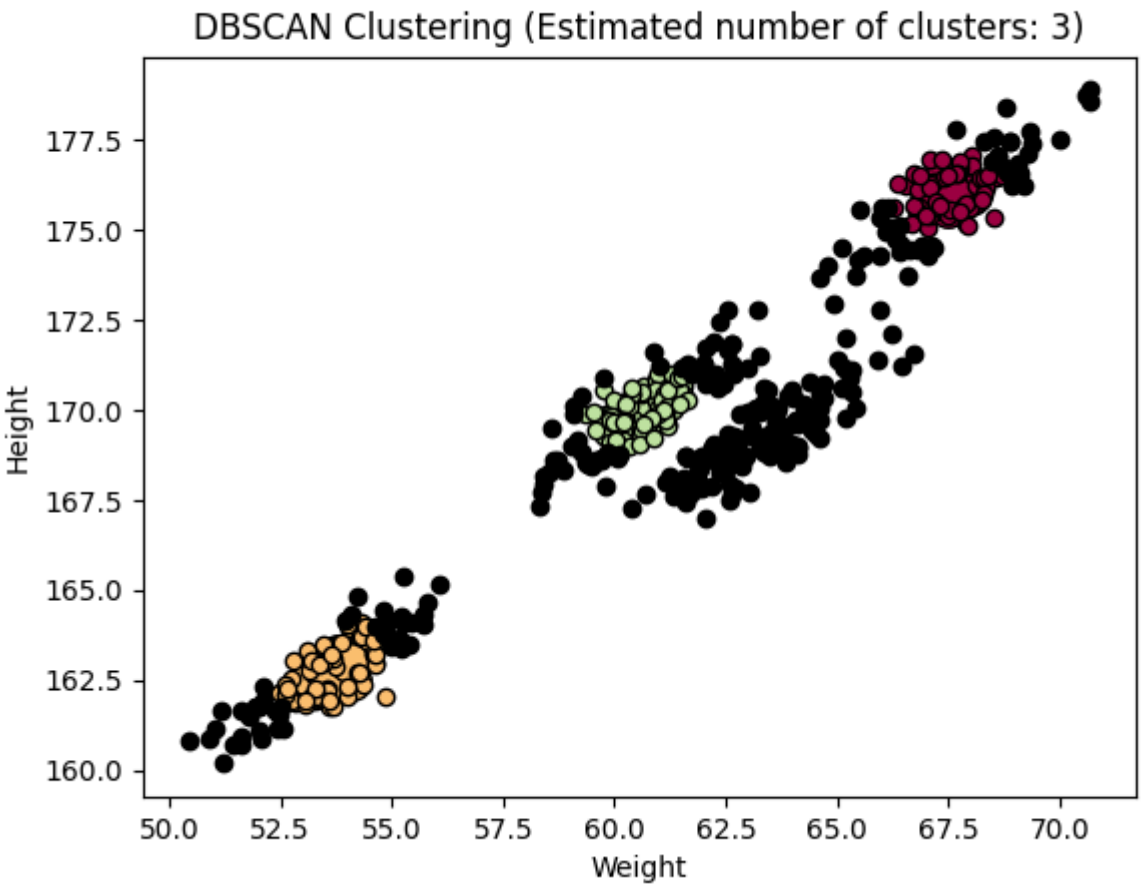


eps = 0.9 again but min_samples = 20



> As you can see, I changed min_samples from 2 to 20 and as a result outlier points have increased

eps = 0.9, min_samples = 50

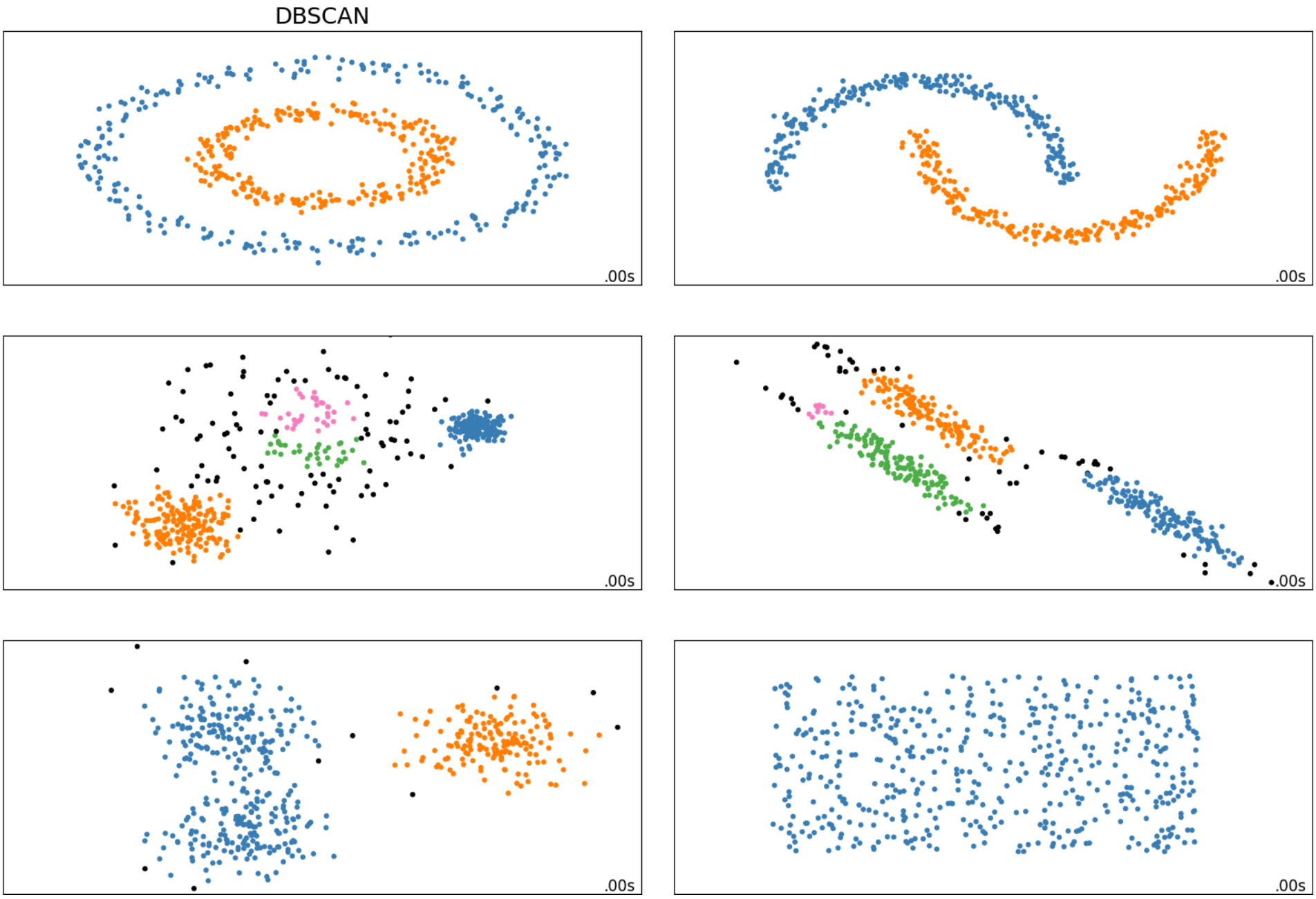


> Again, as I increase the number of min_samples, the number of outlier points is increasing, cause less and less points are in the same radius.

Move to another random dataset

which was generated by:

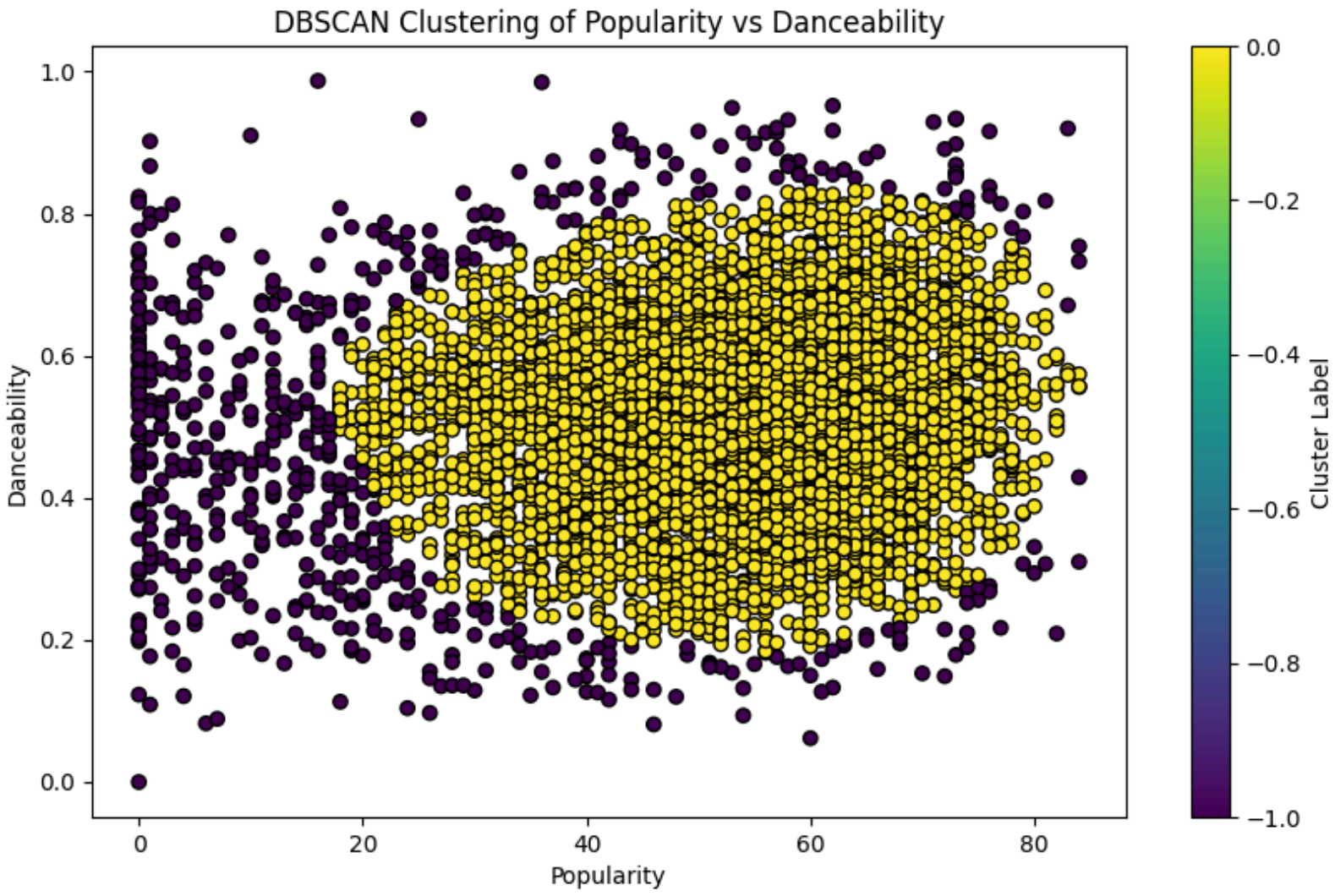
```
from sklearn import cluster, datasets

n_samples = 500
seed = 30
noisy_circles = datasets.make_circles(
    n_samples=n_samples, factor=0.5, noise=0.05, random_state=seed
)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=0.05, random_state=seed)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=seed)
rng = np.random.RandomState(seed)
no_structure = rng.rand(n_samples, 2), None
```



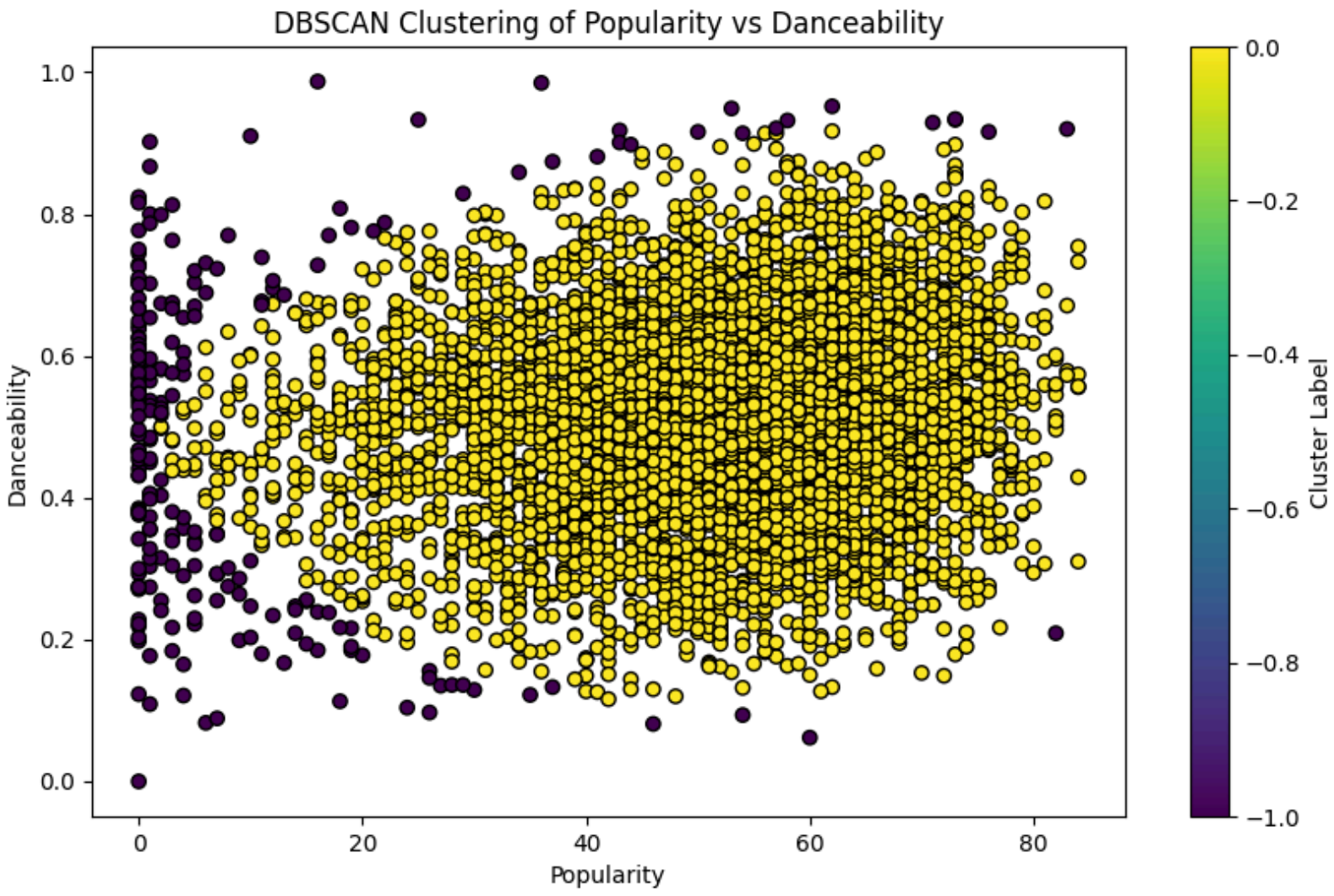
Here you can see, why DBSCAN is powerful in those scenarios

Now get back to our main data about Rock Songs and cluster it using DBSCAN

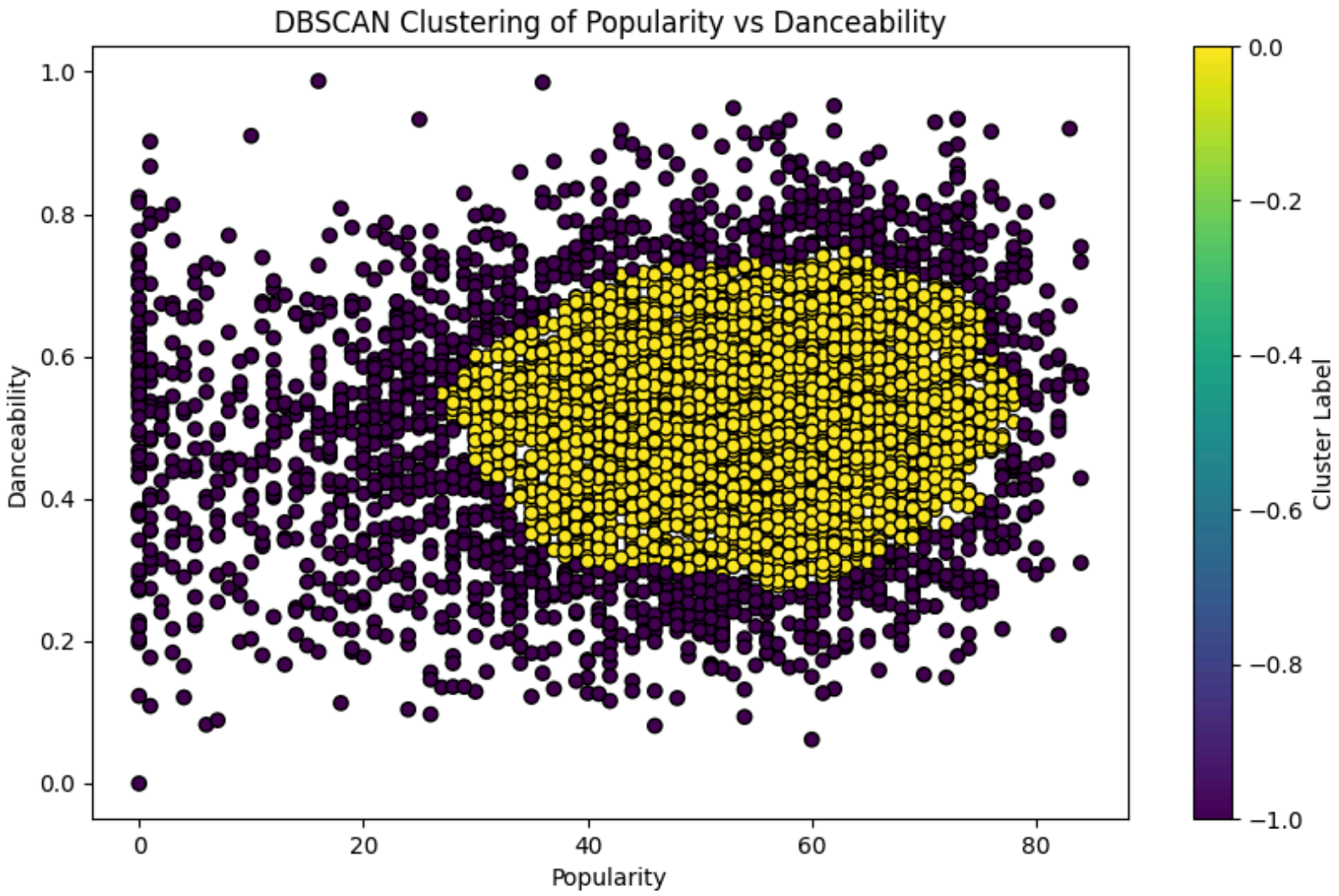
```
df = pd.read_csv('RockData.csv')

df_selected = df[['popularity', 'danceability']].values

scaler = StandardScaler()
df_scaled = scaler.fit_transform(df_selected)

dbscan = DBSCAN(eps=0.5, min_samples=190)
clusters = dbscan.fit_predict(df_scaled)
```



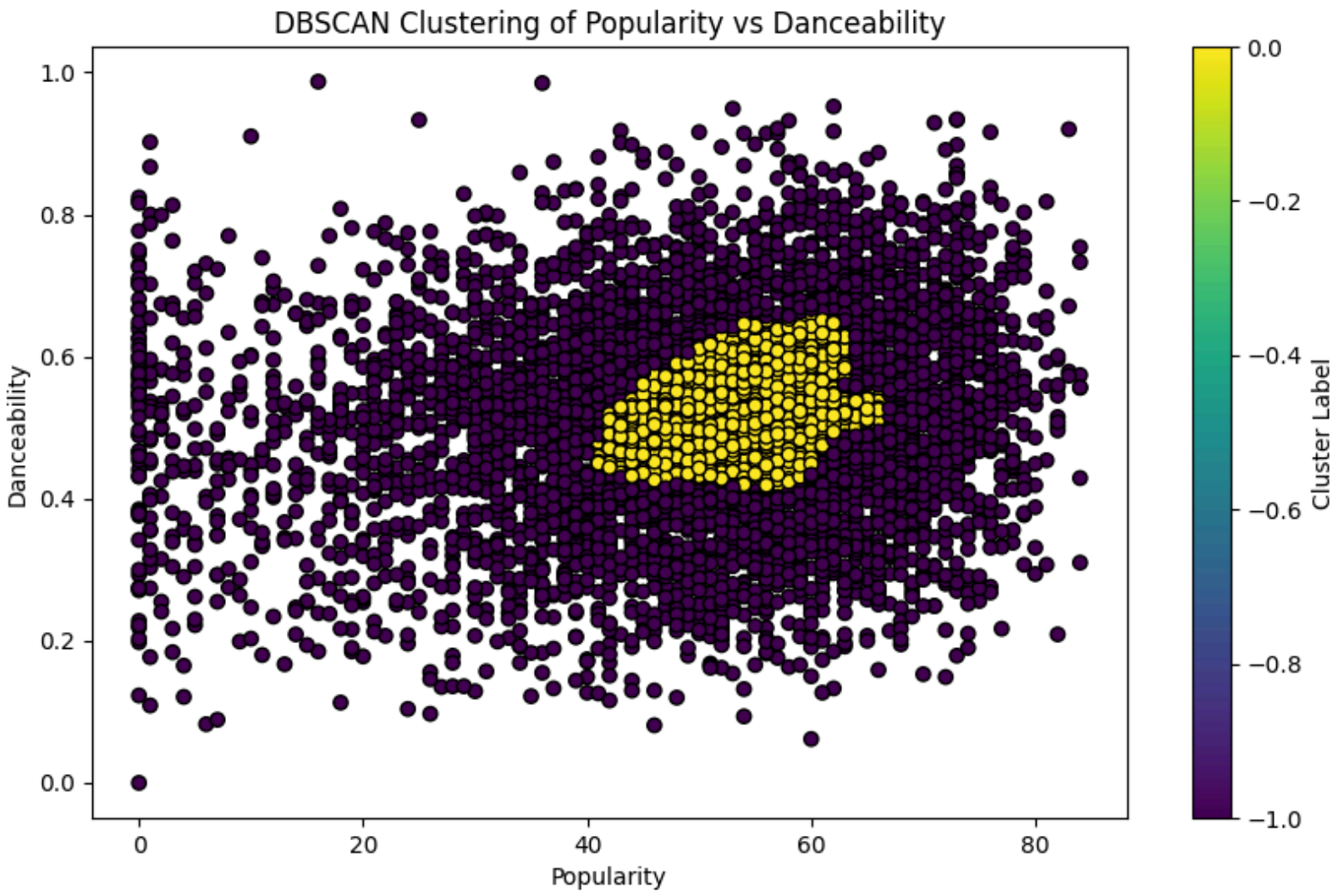
That's how the result looks like



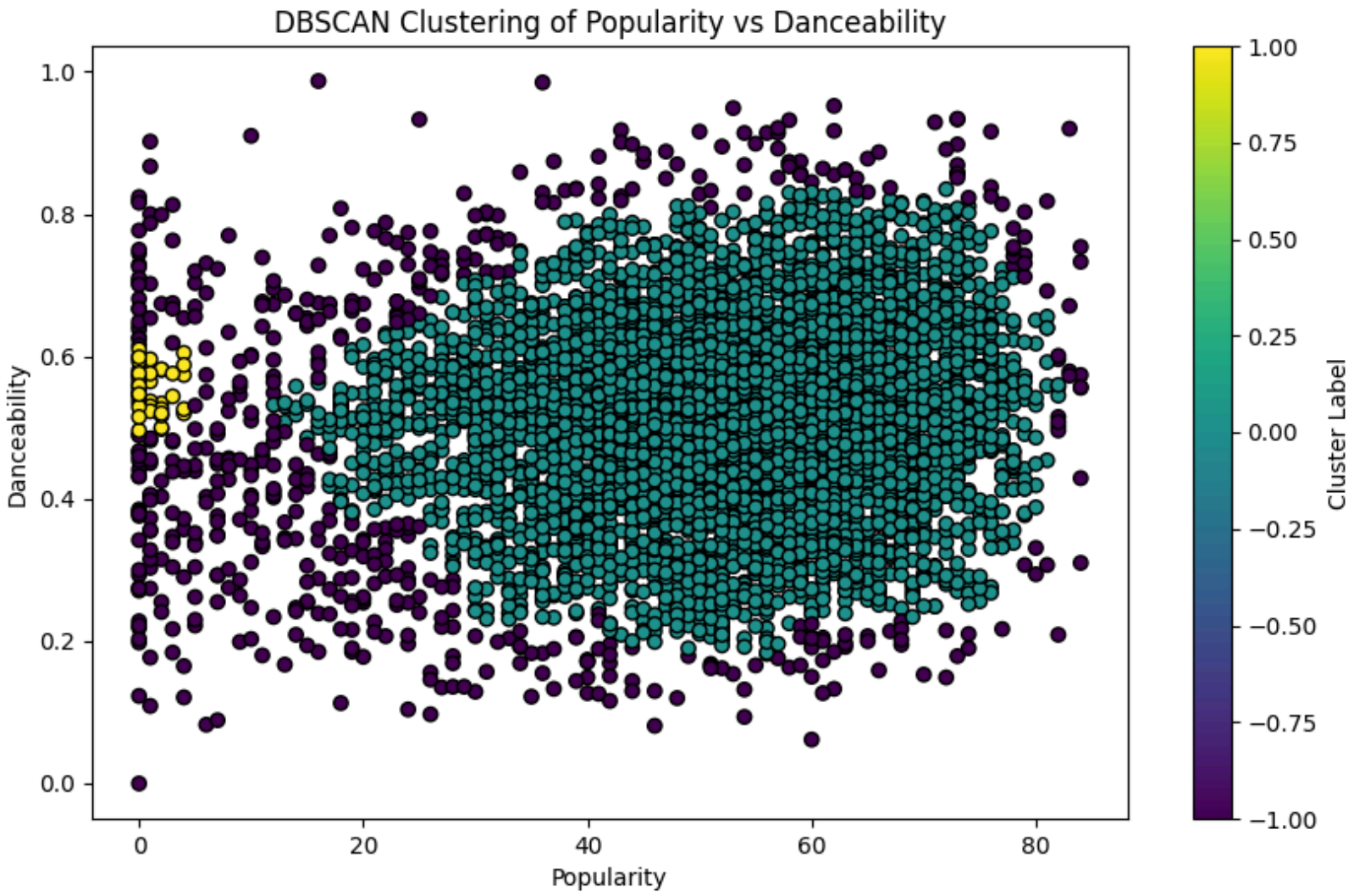

```
dbscan = DBSCAN(eps=0.6, min_samples=120)
clusters = dbscan.fit_predict(df_scaled)
```




```
dbscan = DBSCAN(eps=0.39, min_samples=120)
clusters = dbscan.fit_predict(df_scaled)
```




```
dbscan = DBSCAN(eps=0.21, min_samples=120)
clusters = dbscan.fit_predict(df_scaled)
```




```
dbscan = DBSCAN(eps=0.2, min_samples=20)
clusters = dbscan.fit_predict(df_scaled)
```

You can see the difference as I change epsilon parameter which is the *radius from the core point*

> All the code snippets will be uploaded that I've used here