Coding Challenge

Given a sequence of N-dimensional data, write a program to group these data into multiple clusters. Make reasonable choices to decide how to cluster as well as how many clusters. Discuss any trade-offs involved.

Vinicius Ribeiro da Silva - ribeiro.v.silva@gmail.com

1. Introduction

For this coding challenge, I created a project called **cpp-clustering** to test different approaches to clustering data using the C++ programming language. The project is available in the Git repository: https://github.com/vinicius-r-silva/cpp-clustering.

To address the problem of how to cluster data and determine the number of clusters, I explored two traditional partition-based clustering algorithms: **K-Means** and **DBSCAN**.

K-Means follows these steps:

1. **Initialization**: Randomly initialize *K* centroid points.
2. **Assignment**: Assign each point to its closest centroid.
3. **Update**: Recalculate each centroid as the mean of all points assigned to it.
4. **Iteration**: Repeat steps 2 and 3 until the centroids no longer change.

DBSCAN involves the following steps:

1. **Find core points**: Identify all points in the dataset that have a minimum number of neighbors within a specified distance threshold.
2. **Label core points**: Select a random core point to start a new cluster, adding all its neighbors, their neighbors, and so on. Once the current cluster is fully expanded, repeat the process with another unlabeled core point. Continue until all core points are assigned to clusters.
3. **Label remaining points.** Label all remaining non core points to a cluster if the point is a neighbor of a core point. If the point is not neighbors of any core point, left as an outlier.

2. How to cluster

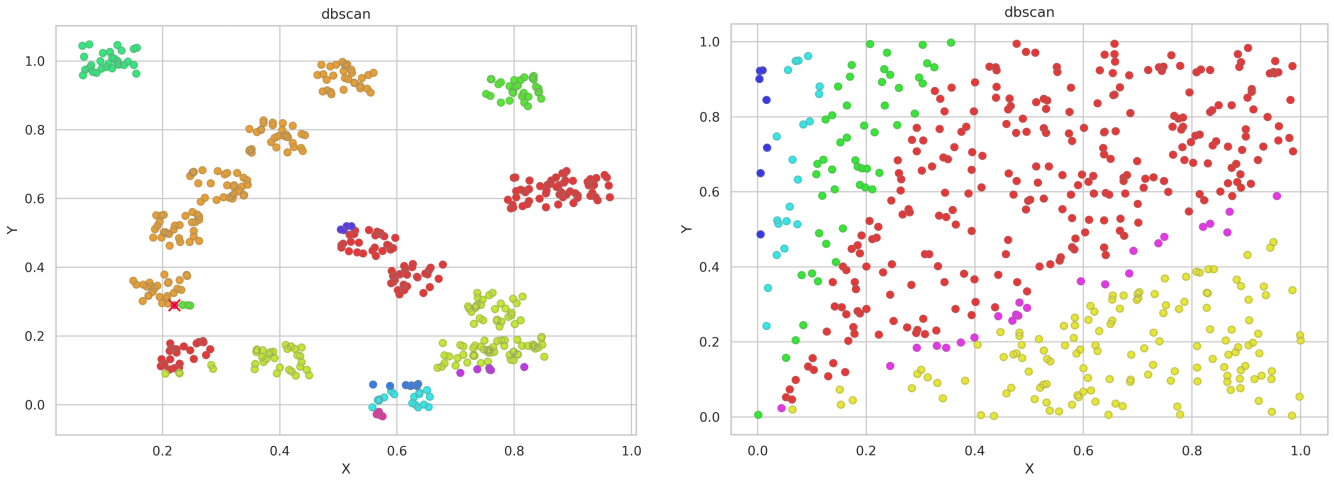
In addition to exploring two different clustering algorithms, a critical part of clusterization is the distance calculation, for that I explored two well known equations, **euclidean distance** and **cosine distance**. These equations are defined as:

$$\text{EuclideanDistance}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

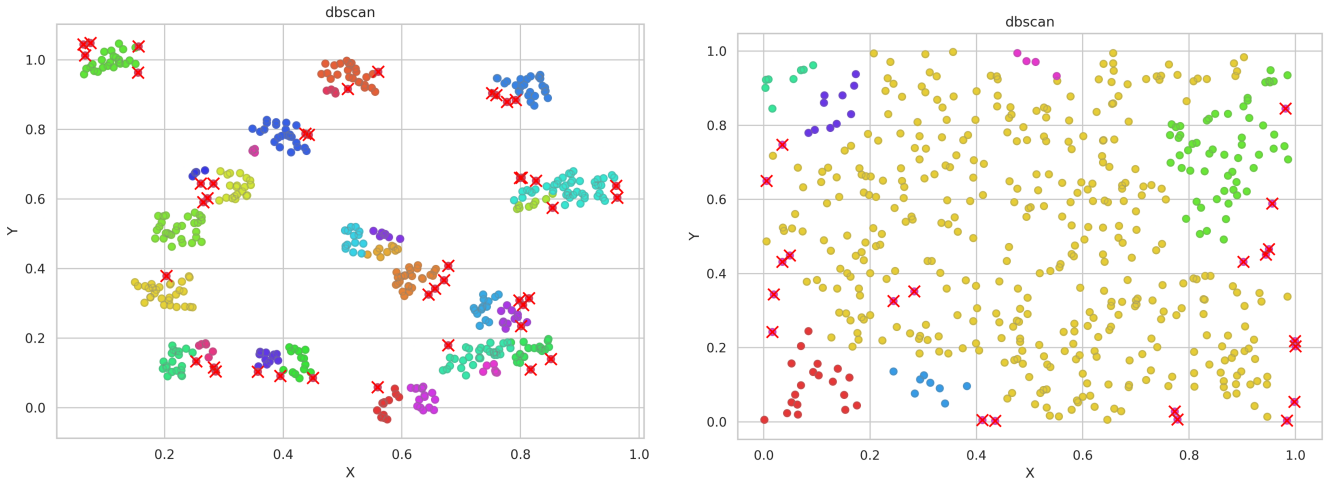
$$\text{CosineSimilarity}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \cdot \sqrt{\sum_{i=1}^{n} y_i^2}}$$

$$\text{CosineDistance}(\mathbf{x}, \mathbf{y}) = 1 - \text{CosineSimilarity}(\mathbf{x}, \mathbf{y})$$

As how the distance equation impacts, it can be observed on the images bellow:



Cluster by Cosine Distance



Cluster by euclidean distance

3. Choosing the Number of Clusters

To determine the number of clusters, I experimented with several approaches. For **DBSCAN**, the number of clusters is implicitly defined by two key parameters:

- **Minimum number of neighbors** required for a point to be considered a core point.
- **Maximum distance** between two points for them to be considered neighbors.

For the **minimum number of neighbors**, I used a constant equal to the number of dimensions in the dataset plus one. To determine the **maximum distance**, I applied the **k-distance graph method**, which involves the following steps:

1. For each point, calculate the distance to its *k*-th nearest neighbor.
2. Collect all these distances into a vector.
3. Sort the vector in ascending order.
4. Identify the "elbow" point (the point of maximum curvature) in the sorted graph and use it as the distance threshold.

For the **K-Means** algorithm, I focused on two main factors:

- **K value calculation.**
- **Centroids selection.**

For the **K value calculation**, I tried two approaches. The first was the **heuristic method** setting K as sqrt(n / 2), where n is the number of data points. The second is using **Kneedle method with WCSS**.

WCSS (Within-Cluster Sum of Squares) is a cluster evaluation metric where its value is calculated with the sum of the distance of every point from the dataset to its cluster centroid. For example, if the distance is calculated as the squared of the euclidean distance, the formula for WCSS is:

$$\mathrm{WCSS} = \sum_{k=1}^{K} \sum_{\mathbf{x}_i \in C_k} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|^2$$

The **Kneedle method** consists of trying a range of K values, in my case from 1 to sqrt(n / 2), calculating the WCSS for each k value cluster, and identifying the value of k where the score point is farthest from the straight line connecting the first and last values.

For the **Centroids selection**, I tried two different approaches: **pure random selection** and the implementation of **K-Means++**. The pure random selection, named in the project as naive K-Means, takes the first random indexes generated as the initial centroids. On the other hand, K-Means++ tries to optimize this selection by using the following steps

1. Choose the first centroid uniformly at random from the data points.
2. For each data point, compute the distance squared to the nearest already chosen centroid:
$$D(\mathbf{x}_i)^2 = \min_{1 \le j \le m} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|^2$$
3. Choose the next centroid from the data points with probability proportional to its distance from the closest centroid
$$P(\mathbf{x}_i) = \frac{D(\mathbf{x}_i)^2}{\sum_{\mathbf{x}_j \in X} D(\mathbf{x}_j)^2}$$
4. **Repeat Steps 2 and 3** until k centroids are chosen.

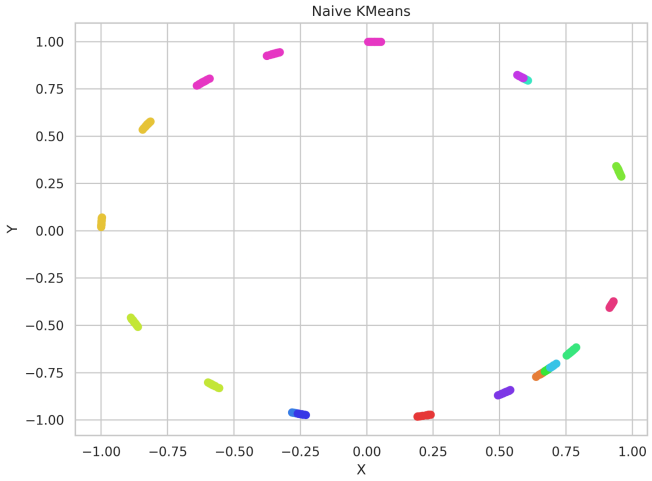
Besides this better initial centroid selection, in the K-Means++ code the algorithm is executed multiple times and in the end is selected the clusters that provide the best WCSS score.

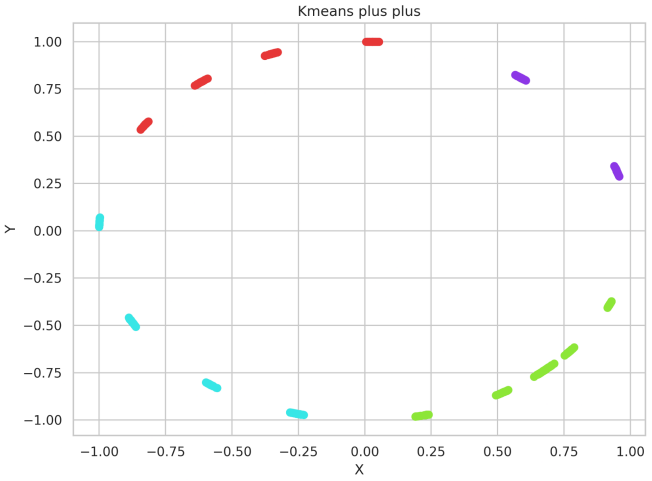
4. Results and discussion

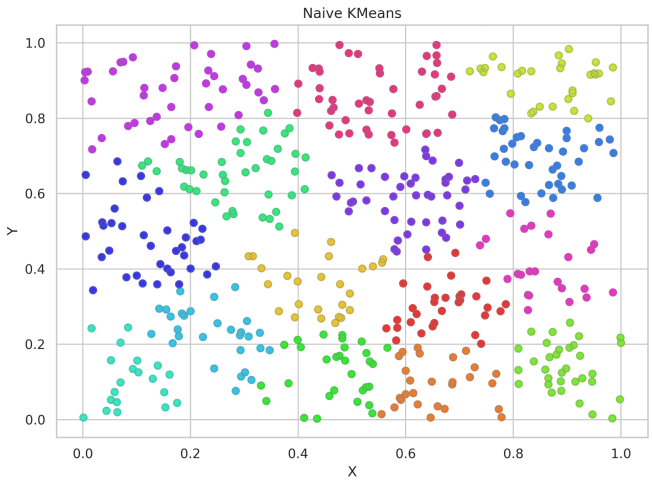
4.1 Naive K-Means and K-Means++

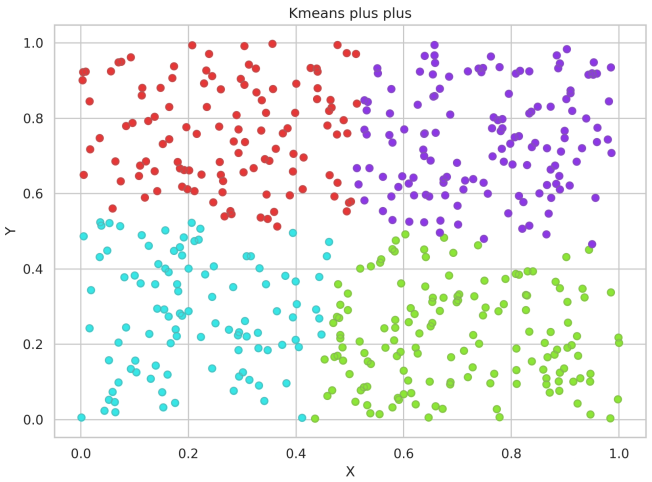
For this comparison, I used a dataset of 500 randomly generated 2D points and applied **Euclidean distance** as the distance metric.

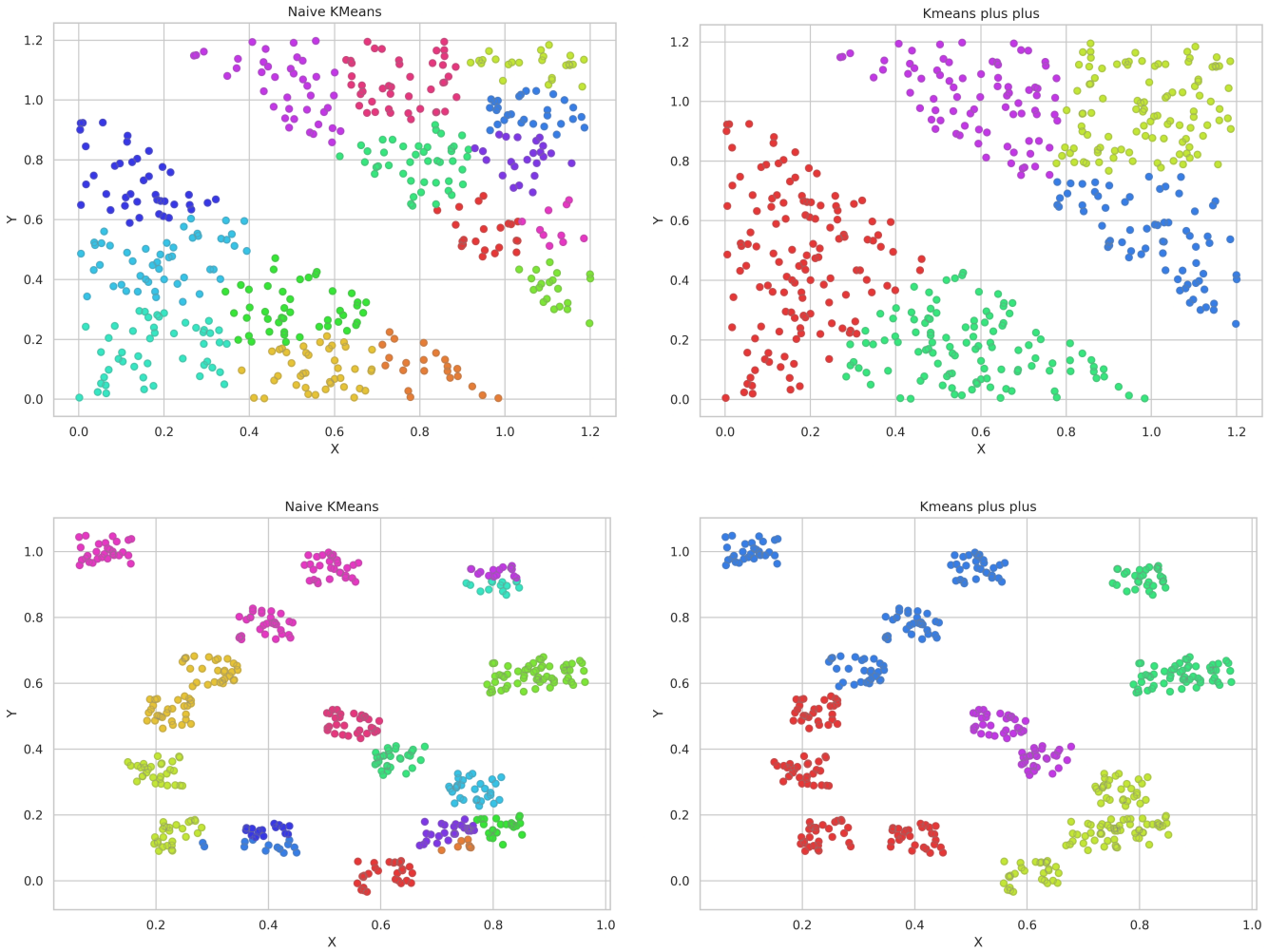
The images below show the clustering results. The images on the **left** correspond to the **naive K-Means** algorithm, while the images on the **right** show the results from **K-Means++**. **Each color in the graph represents one cluster**, making it easier to visually compare the quality and separation of clusters produced by each method.











By analyzing the resulting images, it is clear that **K-Means++** outperforms the naive K-Means algorithm in both the **selection of the number of clusters** and the **overall quality of the clustering**. The clusters produced by K-Means++ are more coherent and better separated, leading to a more accurate representation of the underlying data structure.

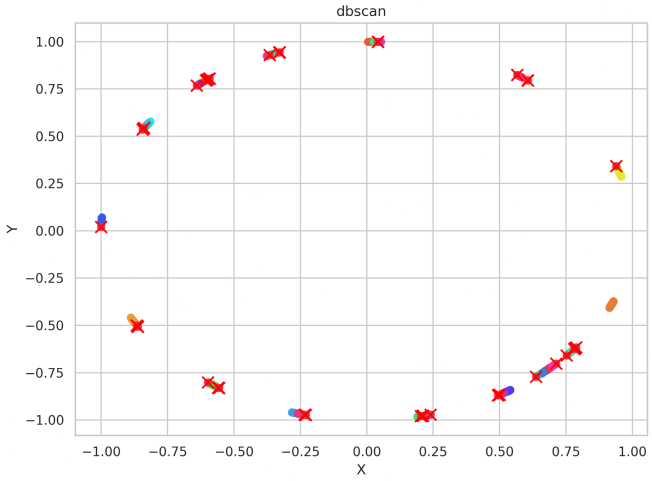
4.2 K-Means++ and DBSCAN

Using the same parameters as before, the following images show clustering results on datasets of 500 two-dimensional points, with **Euclidean distance** used as the distance metric.
Additionally, I calculated the **Silhouette Score** for each clustering result. This score provides a measure of how well each point fits within its cluster, with **higher values indicating better-defined clusters**. The Silhouette Score is calculated using the following formula:

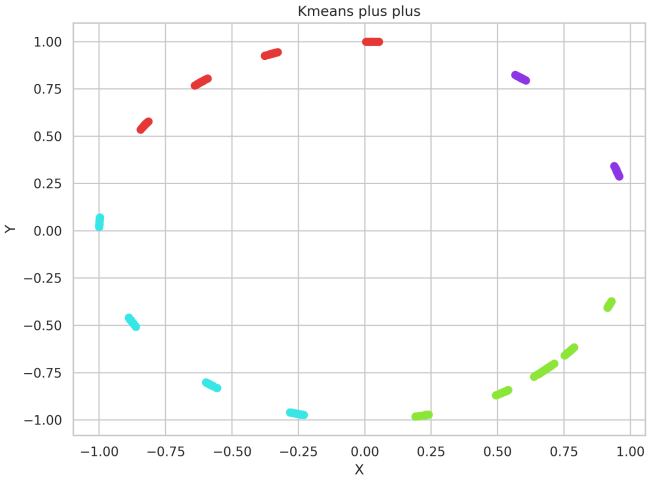
$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

$$S = \frac{1}{n} \sum_{i=1}^{n} s(i)$$

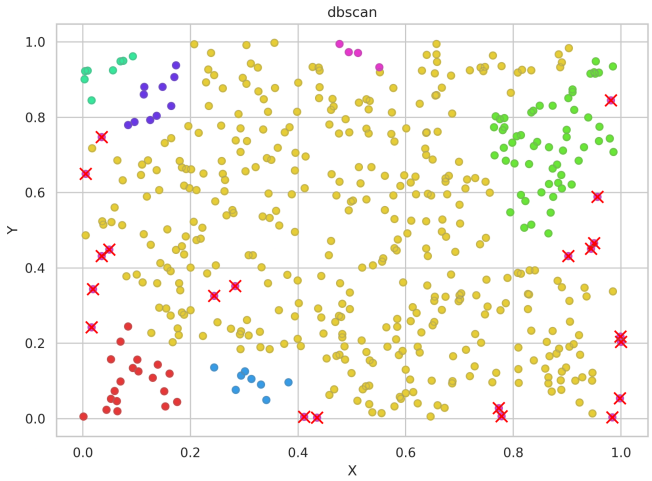
In the visualizations, the images on the **left** correspond to the **DBSCAN** algorithm, while the images on the **right** show the results from **K-Means++**. **Each color represents a different cluster**, while **each "x" symbol represents an outlier** — a point that was not assigned to any cluster.



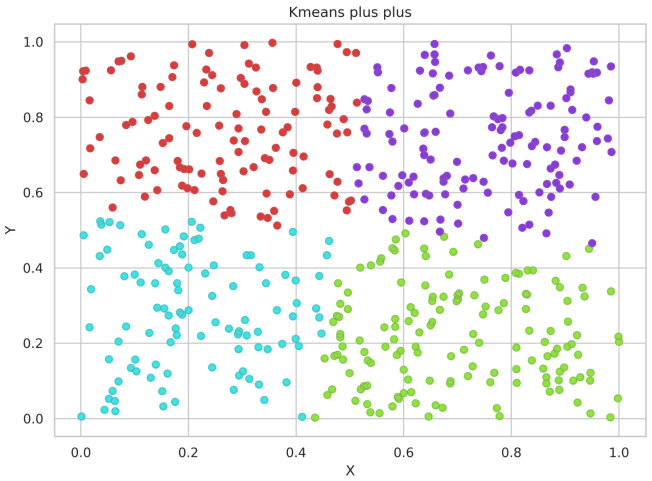
Silhouette Score: 0.931092



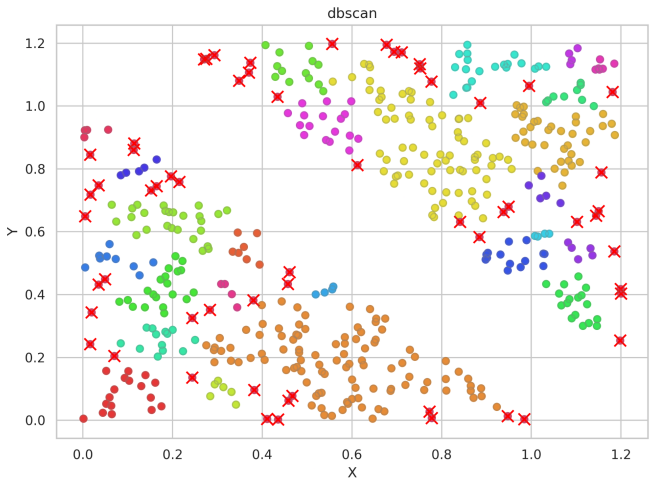
Silhouette Score: 0.640875



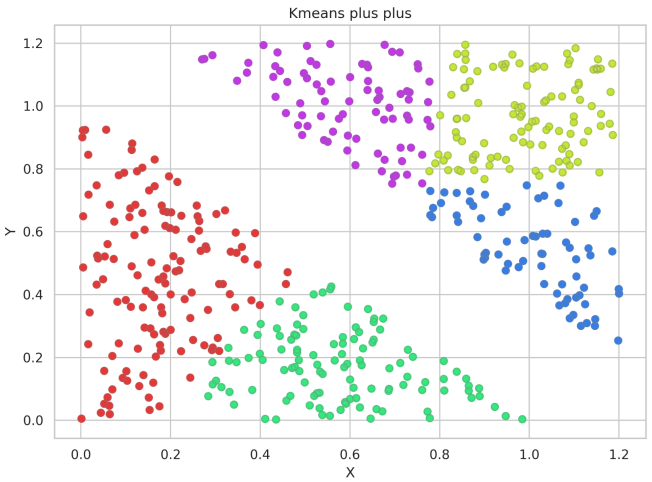
Silhouette Score: -0.138582



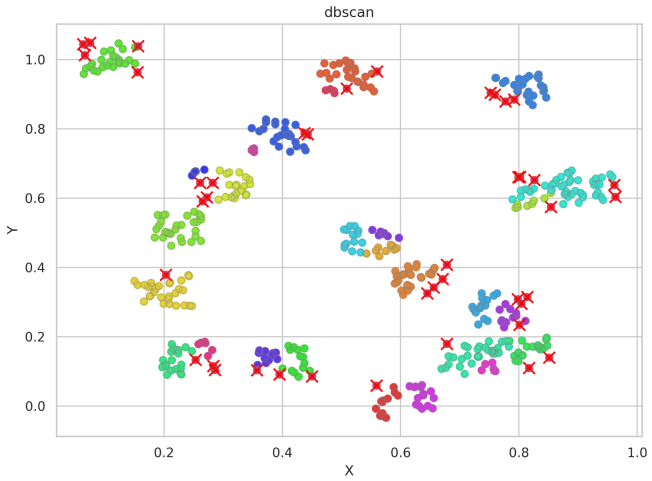
Silhouette Score: 0.421118



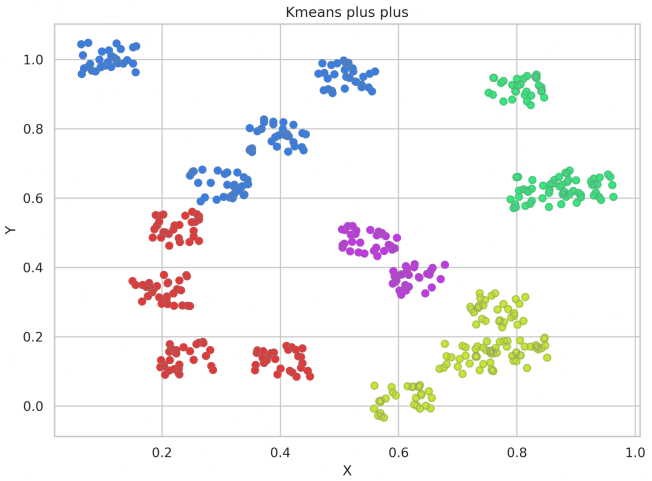
Silhouette Score: 0.0373171



Silhouette Score: 0.429661



Silhouette Score: 0.612126



Silhouette Score: 0.503971

As observed, **DBSCAN** and **K-Means++** results cannot be generalized as one being better than the other. Depending on the generated input, DBSCAN has worse or better silhouette score. It also generally has a higher amount of clusters created, probably because of the input being inherently random and not real data.

DBSCAN has a significant presence of outliers, demonstrating that the minimal distance and minimal amount of neighbors parameters can still be improved.