Hierarchical Clustering & DBSCAN | UNDERFIT | OVERFIT

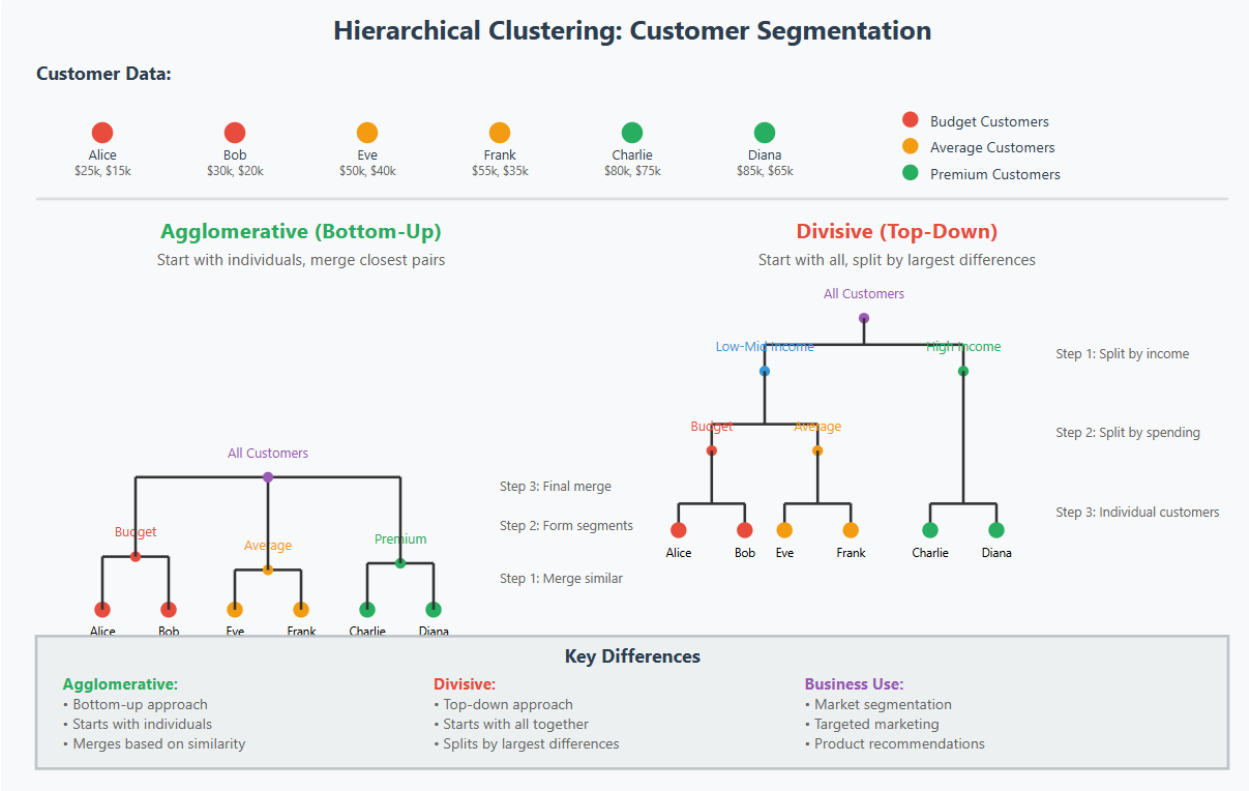


image.png

Hierarchical clustering is used to group similar data points together based on their similarity creating a **hierarchy or tree-like structure**. The key idea is to begin with each data point as its own separate cluster and then progressively merge or split them based on their similarity. Lets understand this with the help of an example

> *Imagine you have four fruits with different weights: an **apple (100g)**, **a banana (120g)**, **a cherry (50g) and a grape (30g)**. Hierarchical clustering starts by treating each **fruit as its own group**.*

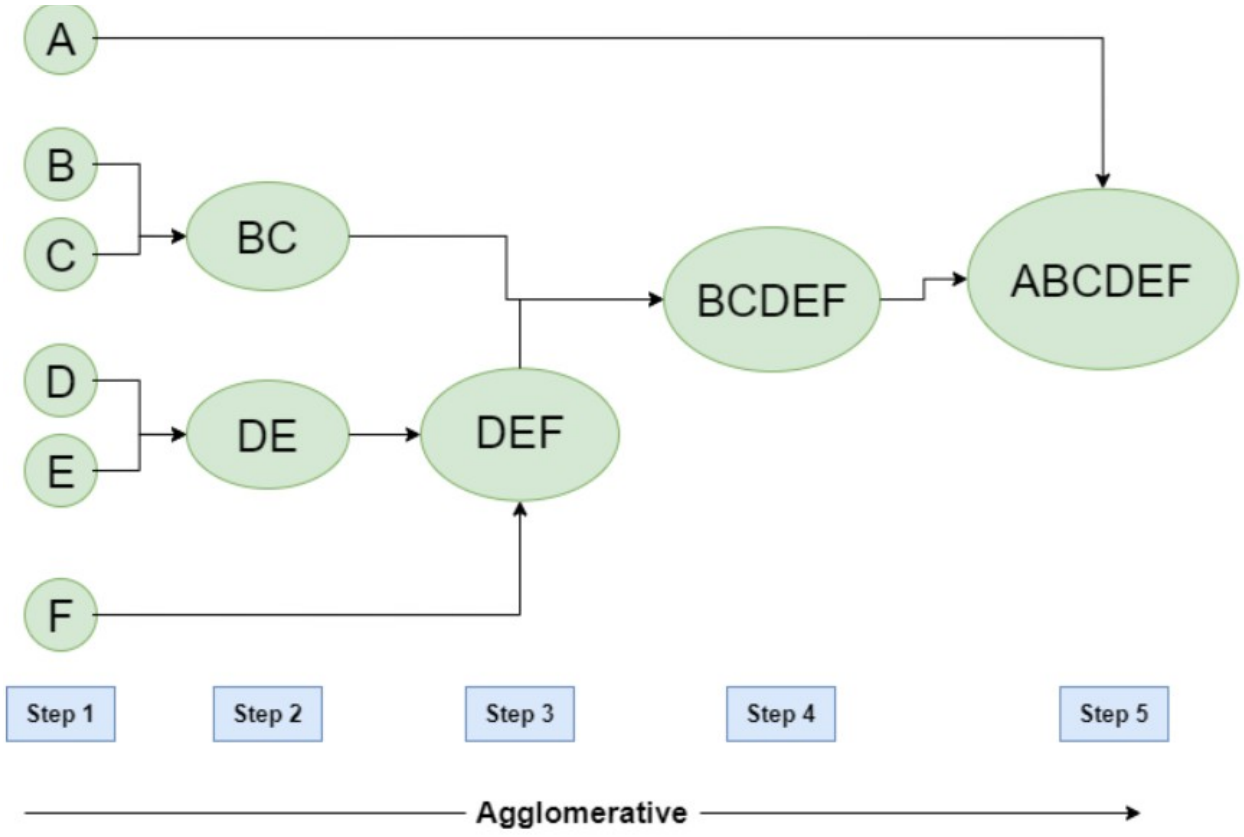
- It then merges the closest groups based on their weights.
- First the cherry and grape are grouped together because they are the lightest.
- Next the apple and banana are grouped together.

 Theory: Hierarchical Clustering

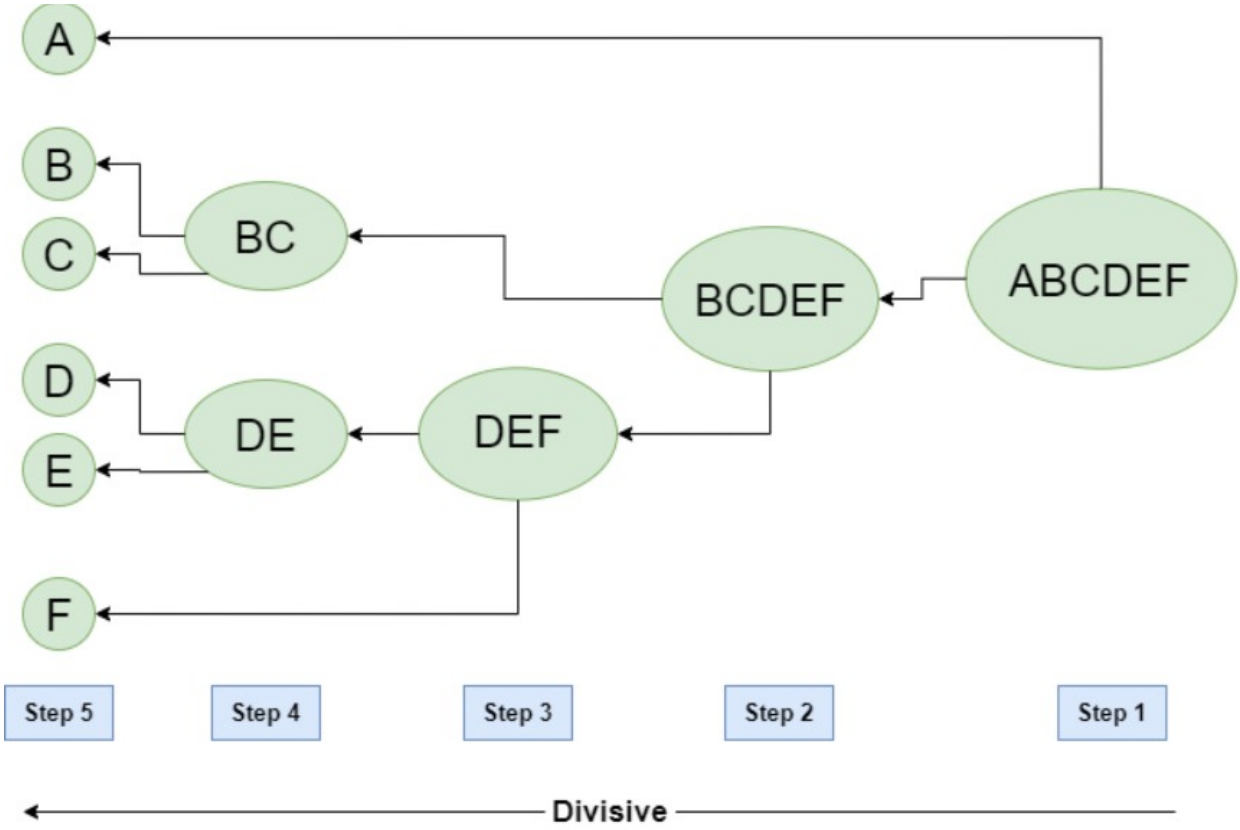
Hierarchical Clustering builds a tree (dendrogram) of clusters without needing to specify the number of clusters.

Two main types:

- **Agglomerative** (bottom-up): Start with each point as its own cluster, merge closest pairs



- **Divisive** (top-down): Start with one big cluster, then split it recursively



Dendrograms visualize this hierarchy and help you decide the number of clusters.

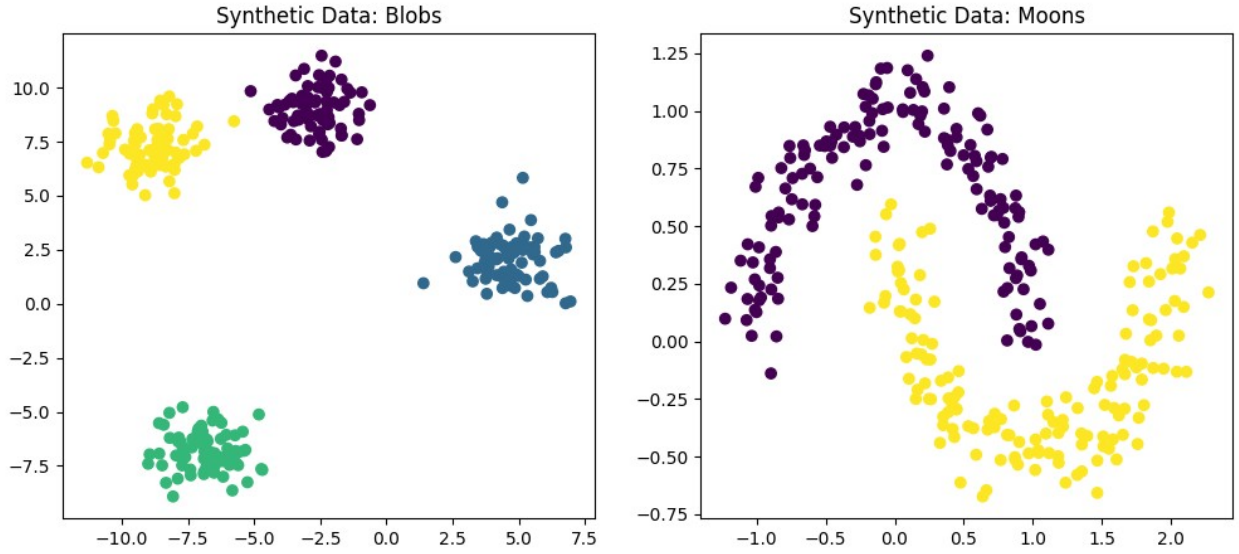
```
# 1. 📦 Import libraries and generate synthetic data
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import make_blobs, make_moons
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN, KMeans
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster

# Generate blob data (good for hierarchical)
X_blob, y_blob = make_blobs(n_samples=300, centers=4, cluster_std=1.0,
random_state=42)
```

`make_blobs: This function from scikit-learn's datasets module is used to create isotropic Gaussian blobs for clustering.`

```
n_samples=300: This specifies that the dataset should contain 300 data points.
centers=4: This indicates that there should be 4 distinct centers for the blobs, effectively creating 4 clusters.
cluster_std=1.0: This sets the standard deviation of the clusters. A value of 1.0 means the points within each cluster are relatively spread out.
random_state=42: This ensures that the data generation is reproducible. If you run the code again with the same random_state, you will get the exact same dataset.
X_blob, y_blob: The function returns two arrays: X_blob contains the features (the x and y coordinates of the points), and y_blob contains the true cluster labels (0, 1, 2, or 3) for each point.

# Generate moon data (good for DBSCAN)
X_moon, y_moon = make_moons(n_samples=300, noise=0.1, random_state=42)
```

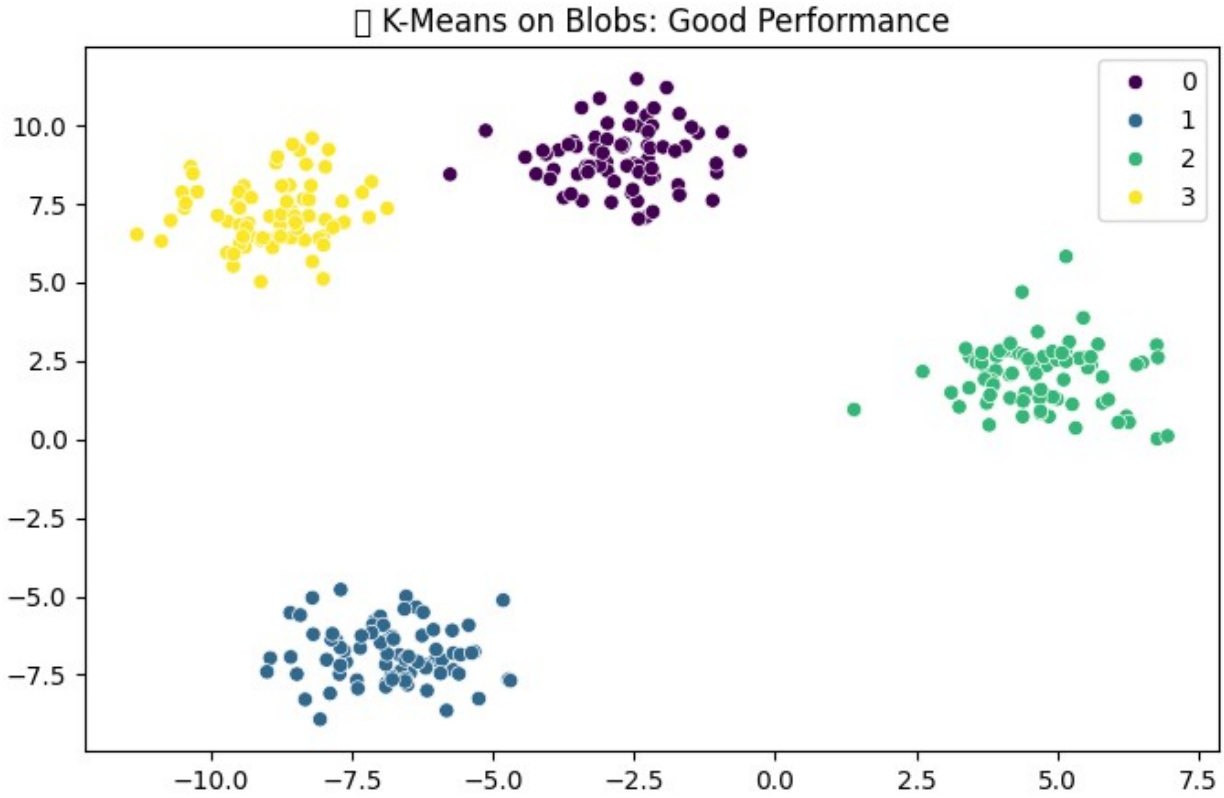
make_moons: This is a function from scikit-learn's datasets module used to create this specific type of non-linearly separable data. n_samples=300: This specifies that the dataset should contain 300 data points. noise=0.1: This adds a small amount of random noise to the data points, making the clusters slightly less perfectly separated and more realistic.

```
# Plot both datasets
fig, axs = plt.subplots(1, 2, figsize=(12, 5))
axs[0].scatter(X_blob[:, 0], X_blob[:, 1], c=y_blob, cmap='viridis')
axs[0].set_title("Synthetic Data: Blobs")
axs[1].scatter(X_moon[:, 0], X_moon[:, 1], c=y_moon, cmap='viridis')
axs[1].set_title("Synthetic Data: Moons")
plt.show()
```



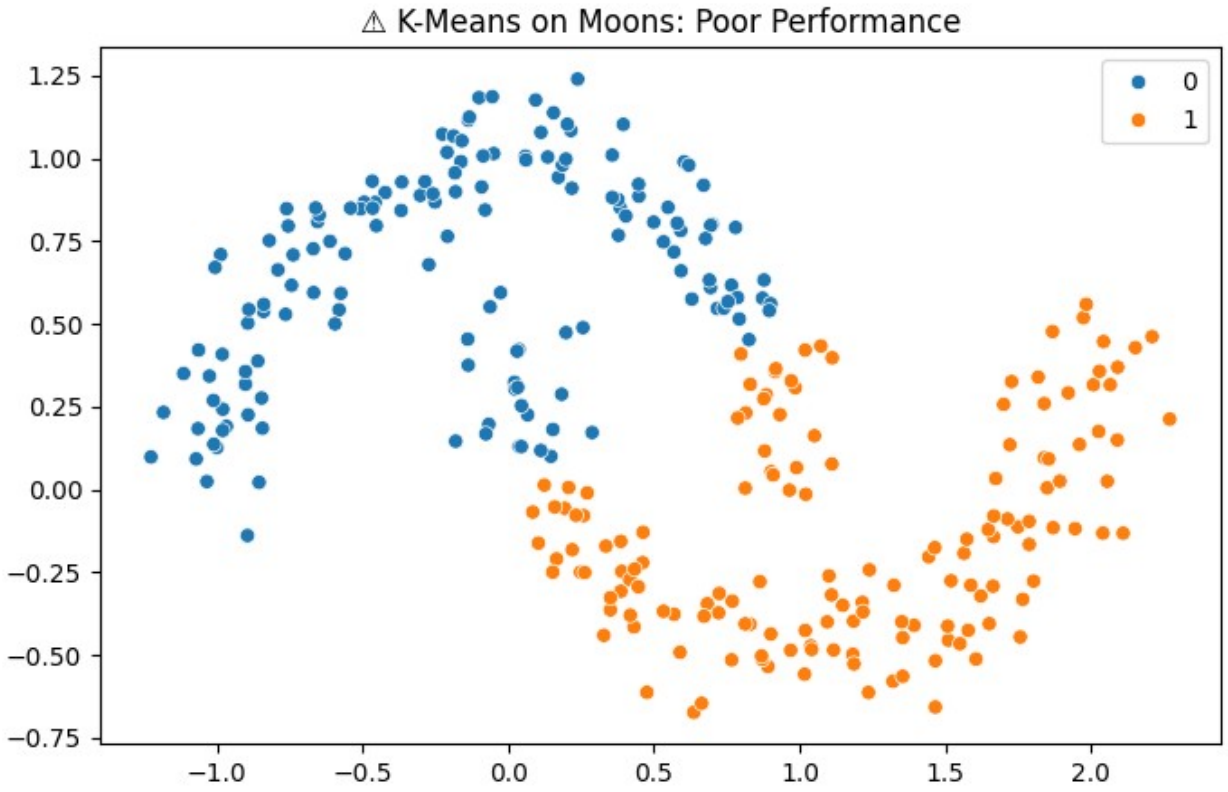
```
# □ KMeans on Blobs
Xb_scaled = StandardScaler().fit_transform(X_blob)
kmeans_blob = KMeans(n_clusters=4, random_state=42) # We know there
are 4 clusters in the blob data
kmeans_blob_labels = kmeans_blob.fit_predict(Xb_scaled)

plt.figure(figsize=(8, 5))
sns.scatterplot(x=X_blob[:, 0], y=X_blob[:, 1],
hue=kmeans_blob_labels, palette='viridis')
plt.title('□ K-Means on Blobs: Good Performance')
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/IPython/core/
pylabtools.py:151: UserWarning: Glyph 9989 (\N{WHITE HEAVY CHECK
MARK}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```

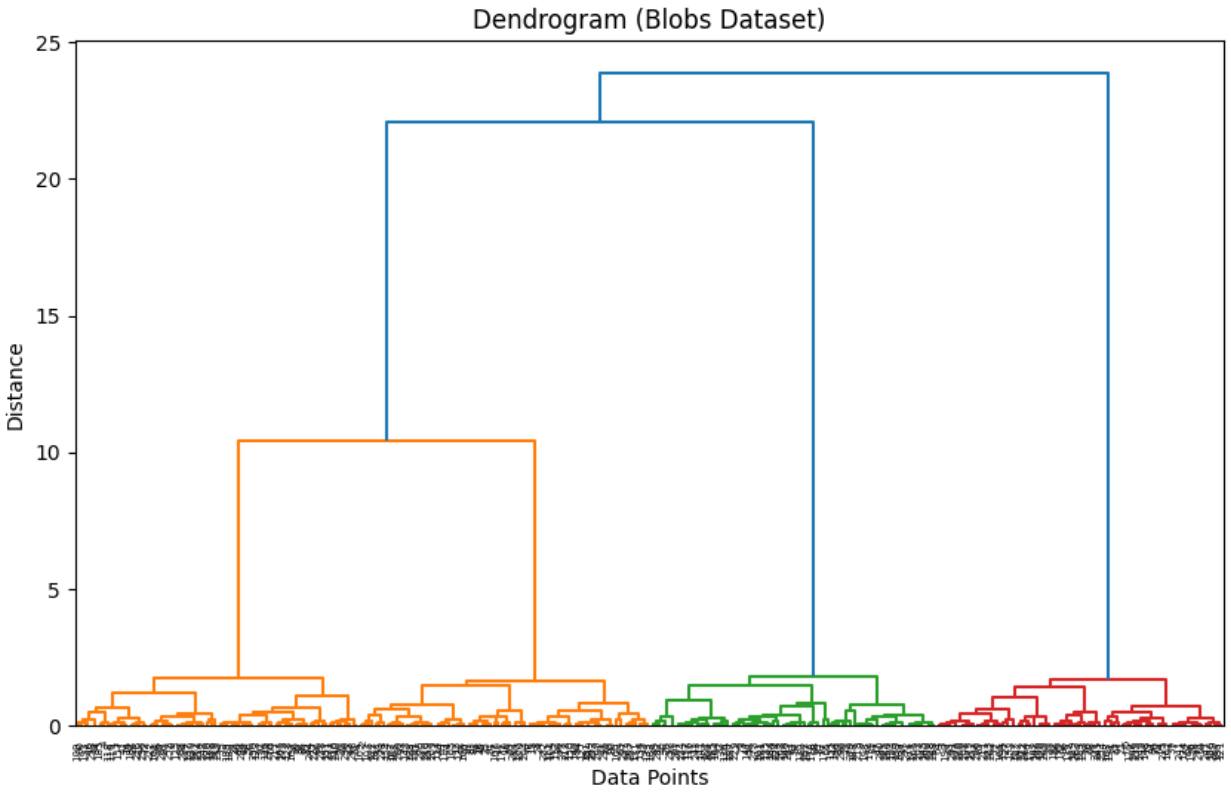



```
# □ KMeans struggles on Moons
Xm_scaled = StandardScaler().fit_transform(X_moon)
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans_labels = kmeans.fit_predict(Xm_scaled)

plt.figure(figsize=(8, 5))
sns.scatterplot(x=X_moon[:, 0], y=X_moon[:, 1], hue=kmeans_labels,
```

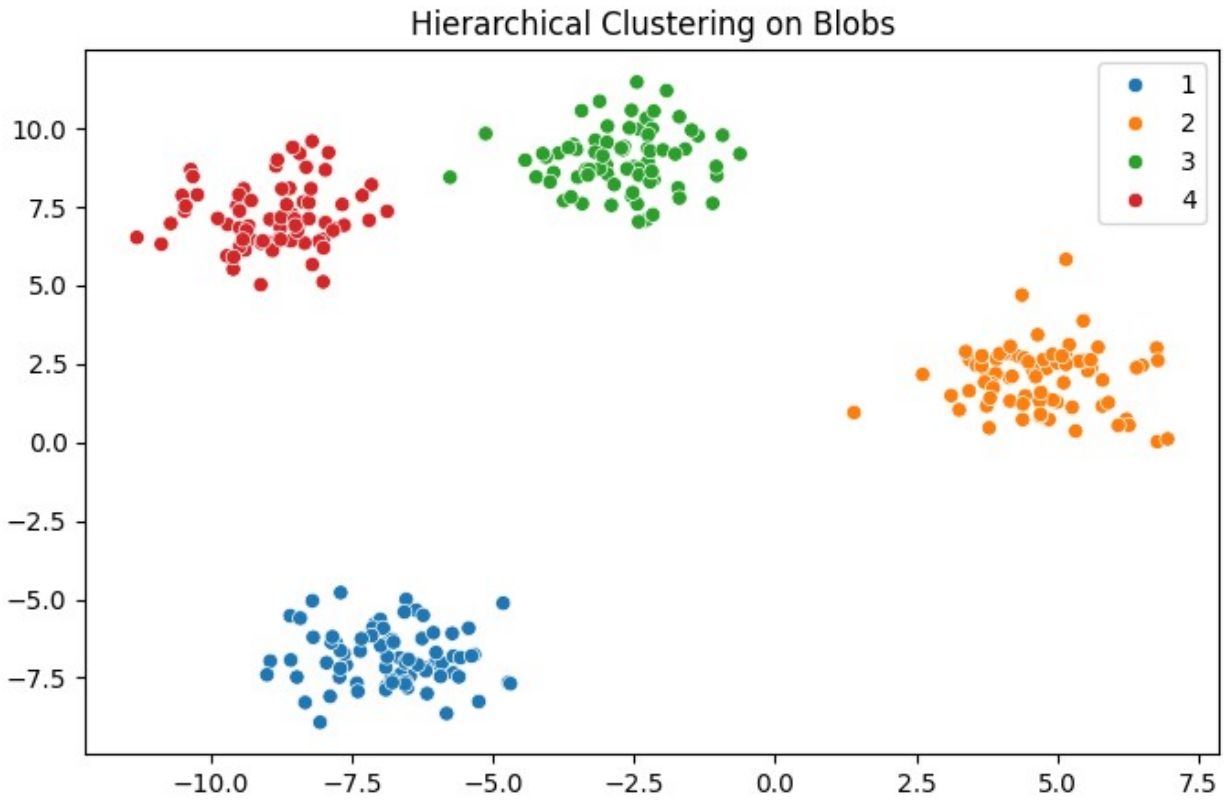
```
palette='tab10')
plt.title('⚠ K-Means on Moons: Poor Performance')
plt.show()
```



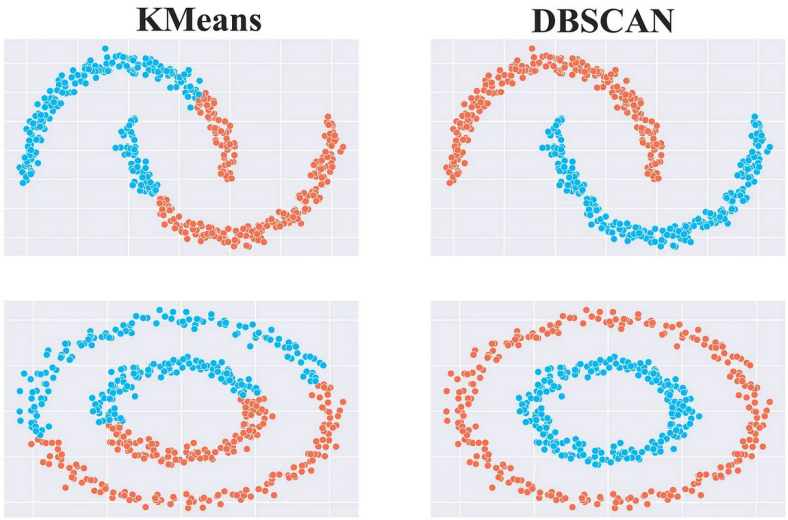

```
# 2. 🌳 Hierarchical Clustering on Blobs
Xb_scaled = StandardScaler().fit_transform(X_blob)
linked = linkage(Xb_scaled,method='ward')

# Plot dendrogram
plt.figure(figsize=(10, 6))
dendrogram(linked, orientation='top', distance_sort='descending',
show_leaf_counts=False)
plt.title('Dendrogram (Blobs Dataset)')
plt.xlabel('Data Points')
plt.ylabel('Distance')
plt.show()
```




```
# Assign clusters
hc_labels = fcluster(linked, t=4, criterion='maxclust')


# Visualize clusters
plt.figure(figsize=(8, 5))
sns.scatterplot(x=X_blob[:, 0], y=X_blob[:, 1], hue=hc_labels,
palette='tab10')
plt.title('Hierarchical Clustering on Blobs')
plt.show()
```



 Theory: DBSCAN (Density-Based Spatial Clustering)



DBSCAN clusters data based on the density of points:

- High-density regions form clusters

- Low-density regions become noise or outliers

Parameters:

- `eps`: distance threshold for neighbors

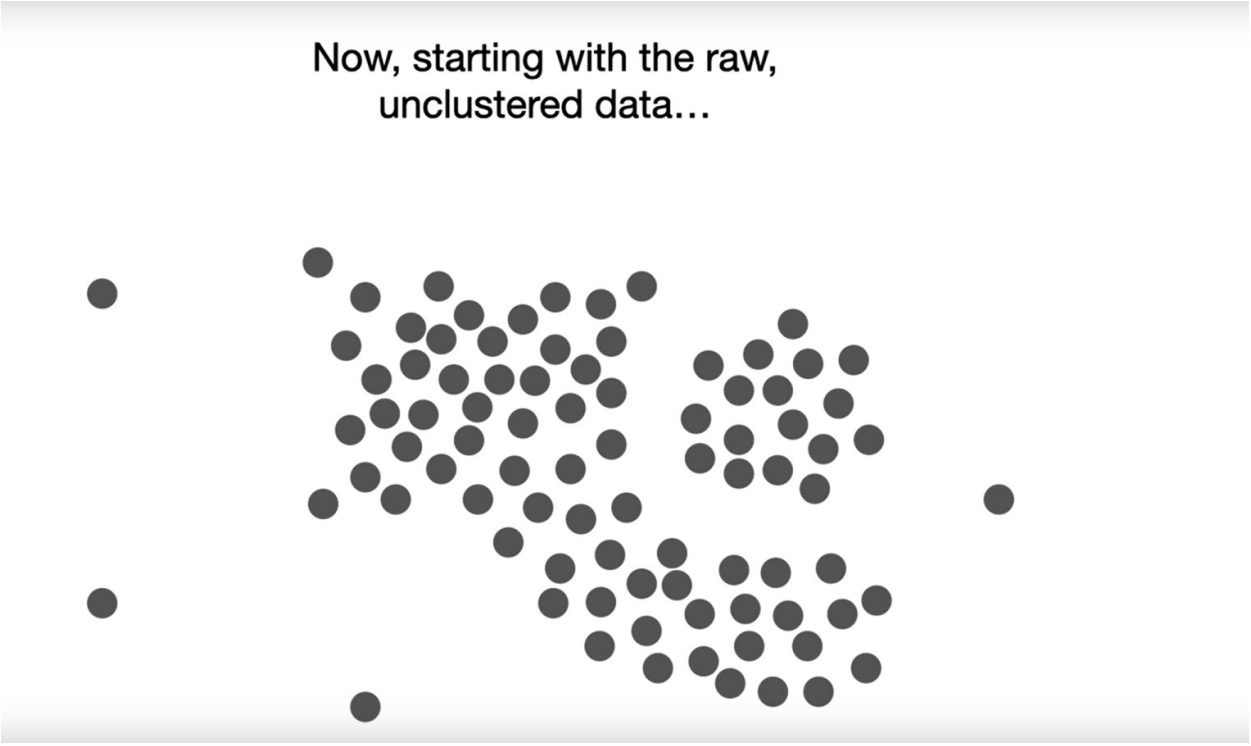
- `min_samples`: minimum number of neighbors to form a dense region

Pros:

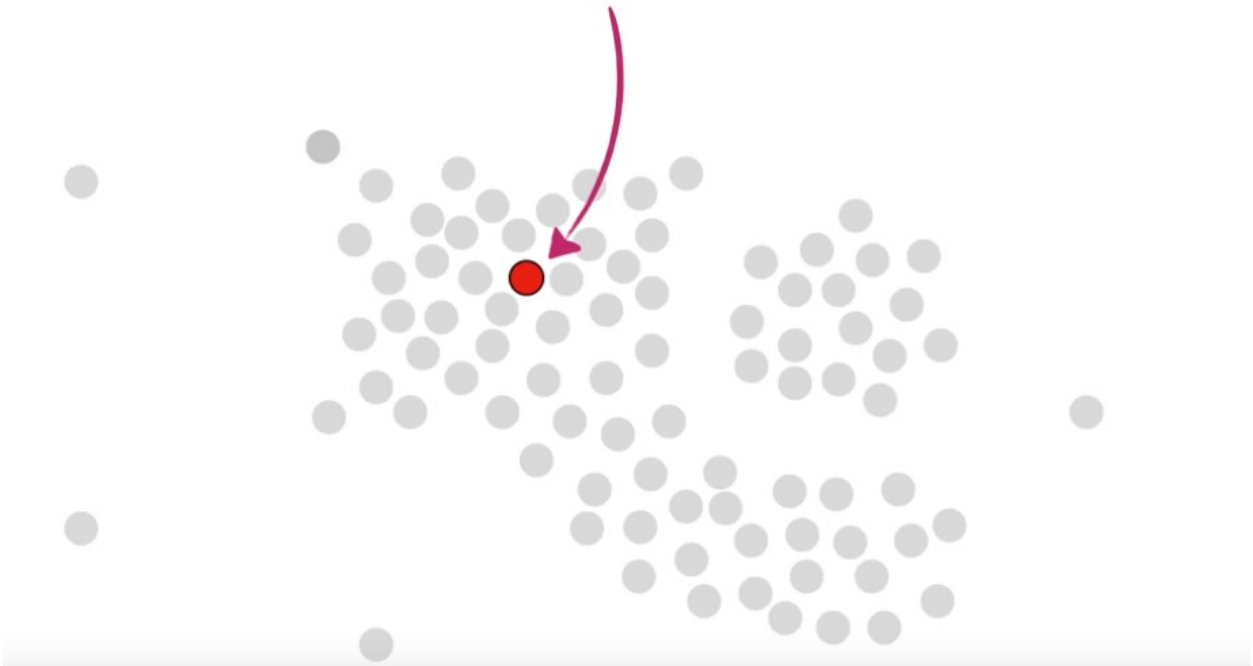
- No need to choose number of clusters

- Can find arbitrarily shaped clusters

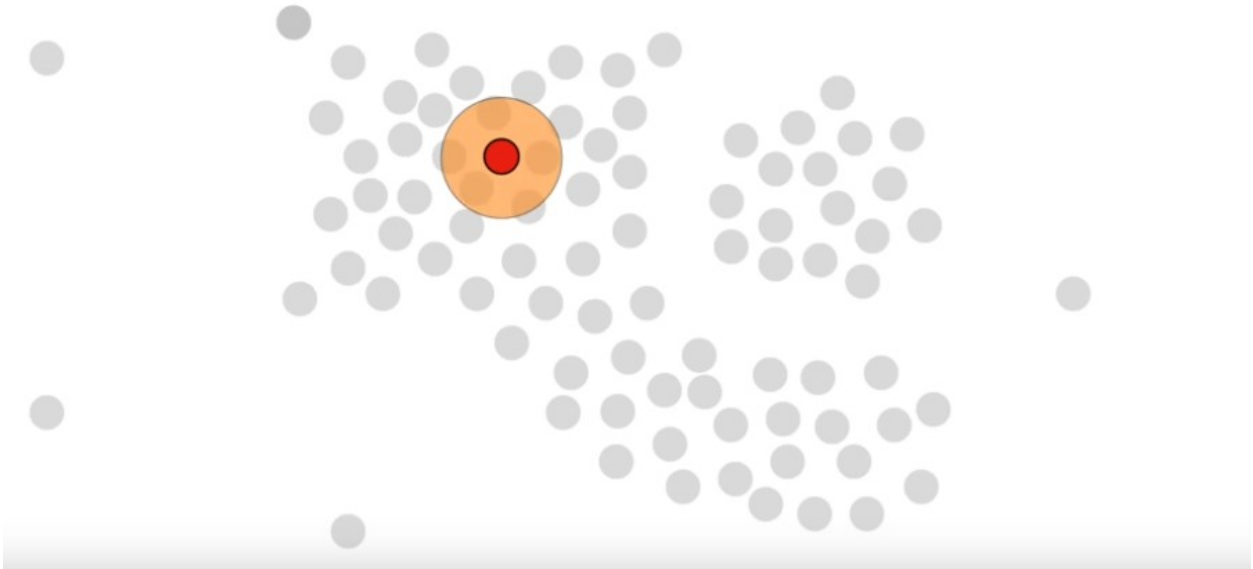
- Identifies noise points



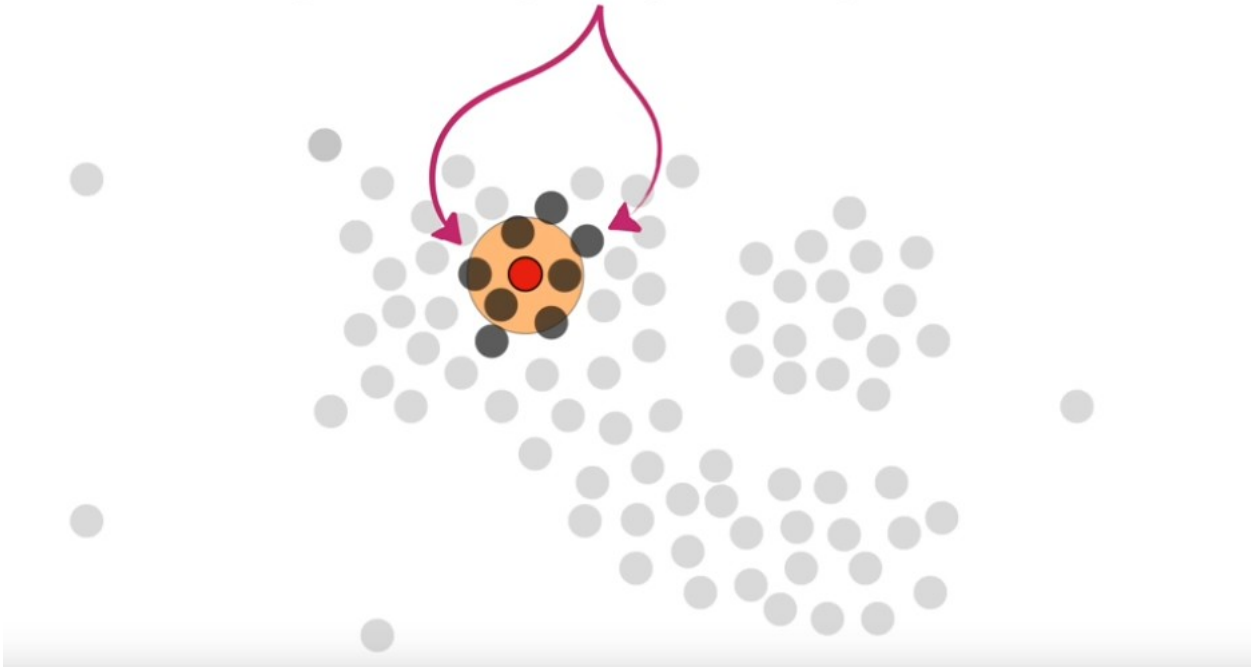
For example, if we start with this **red point**…



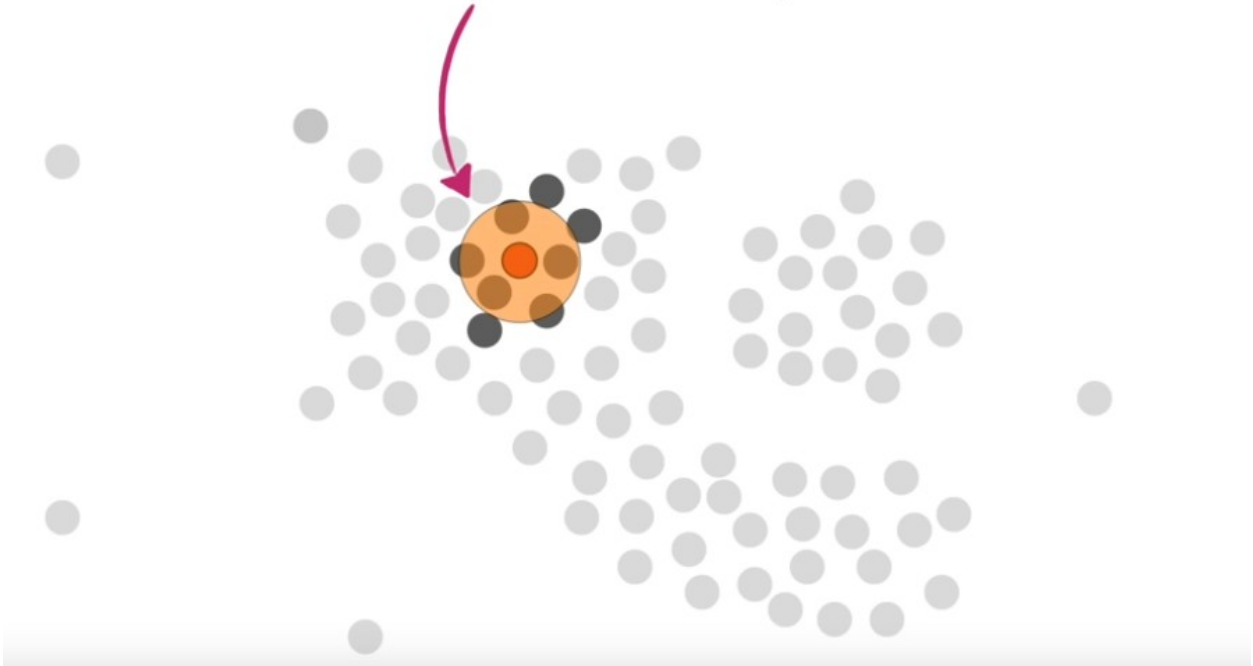
…then we see that the **orange circle** overlaps, at least partially, **8** other points.

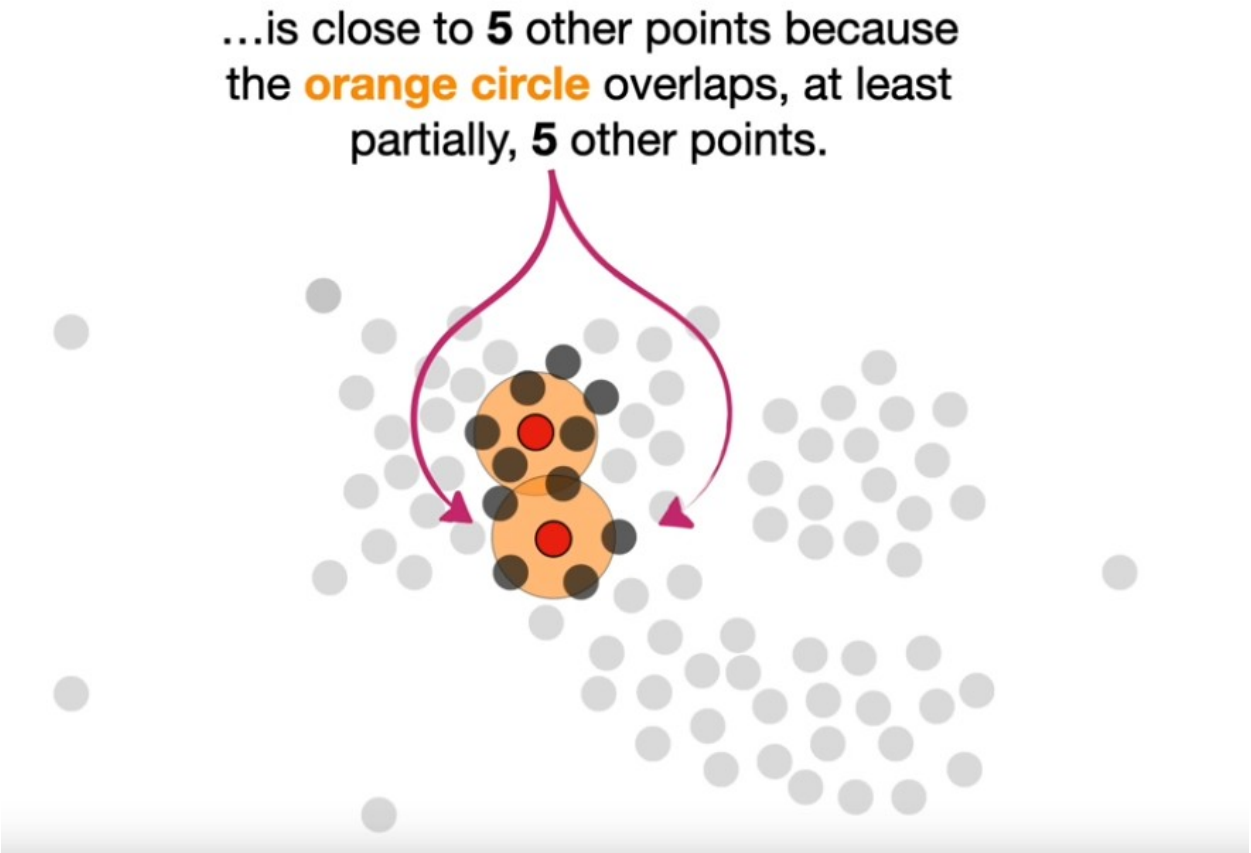


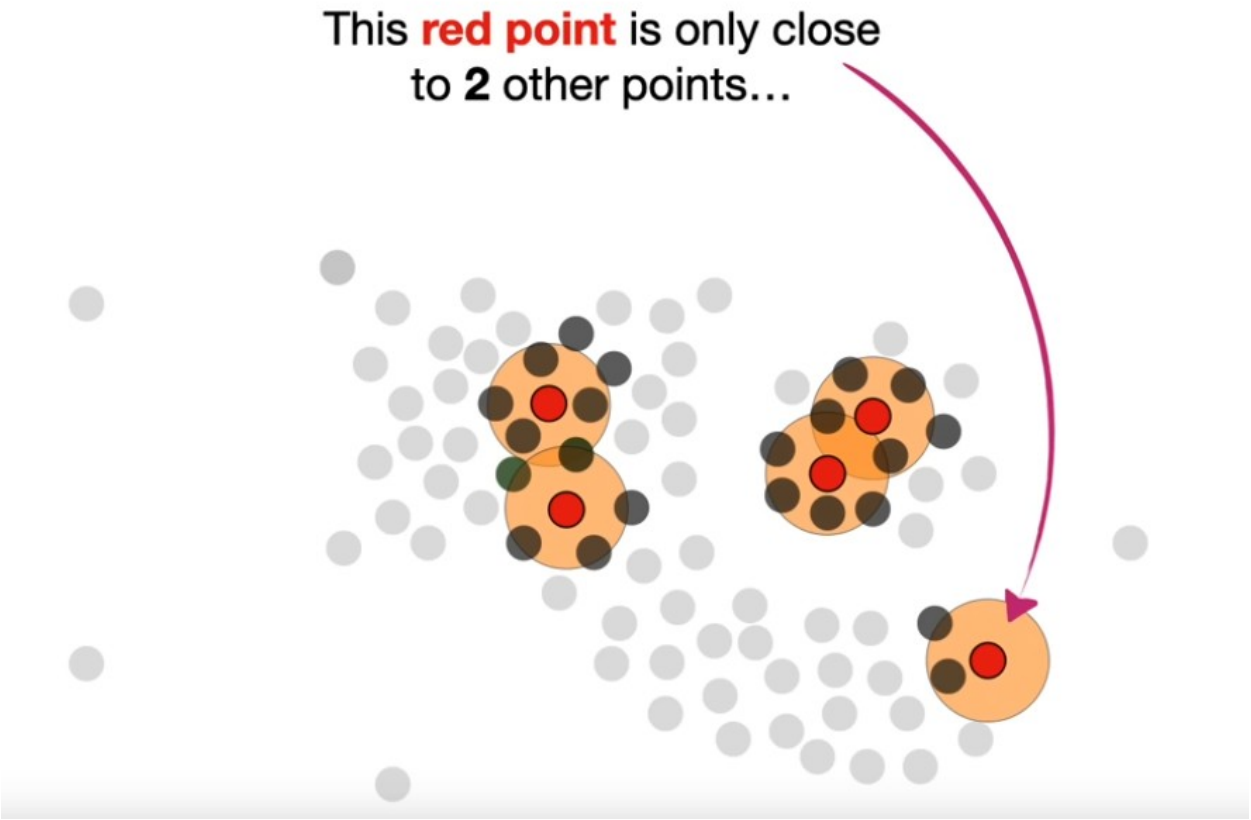
…then we see that the **orange circle** overlaps, at least partially, **8** other points.

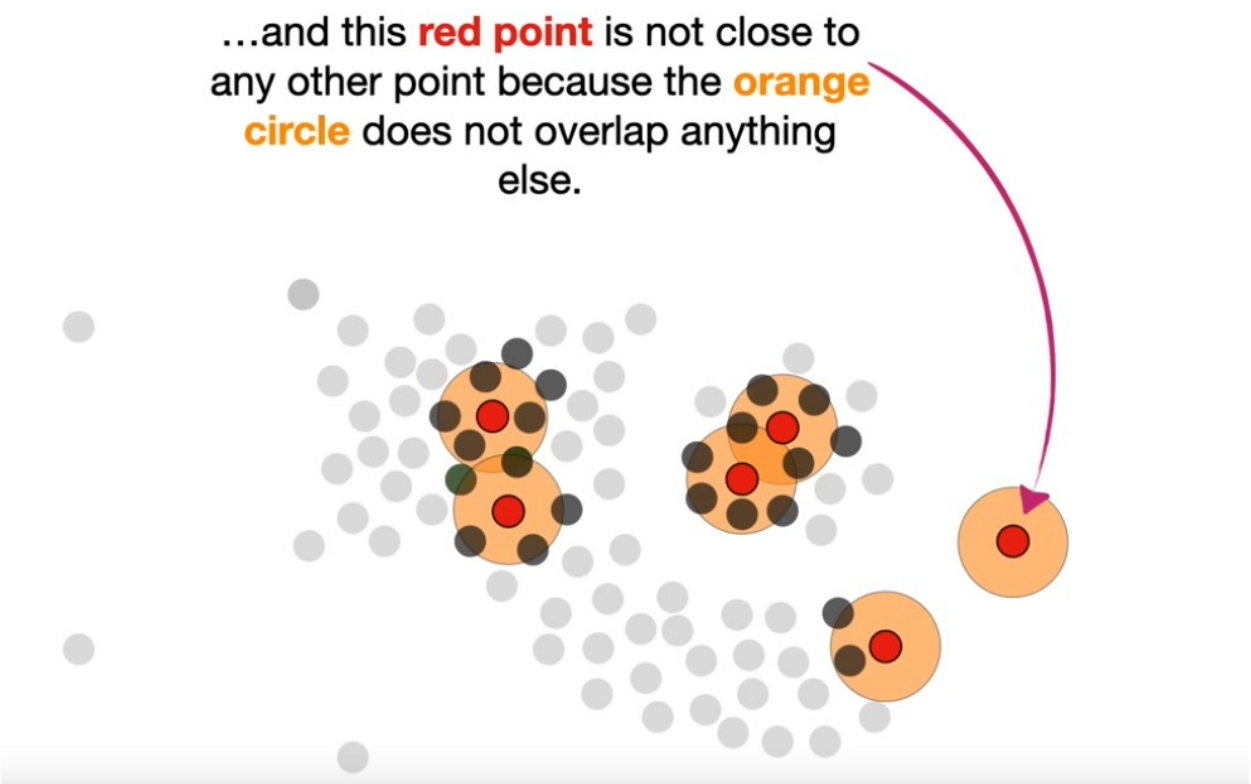


NOTE: The radius of the **orange circle** is user defined, so when using **DBSCAN**, you may need to fiddle around with this parameter.

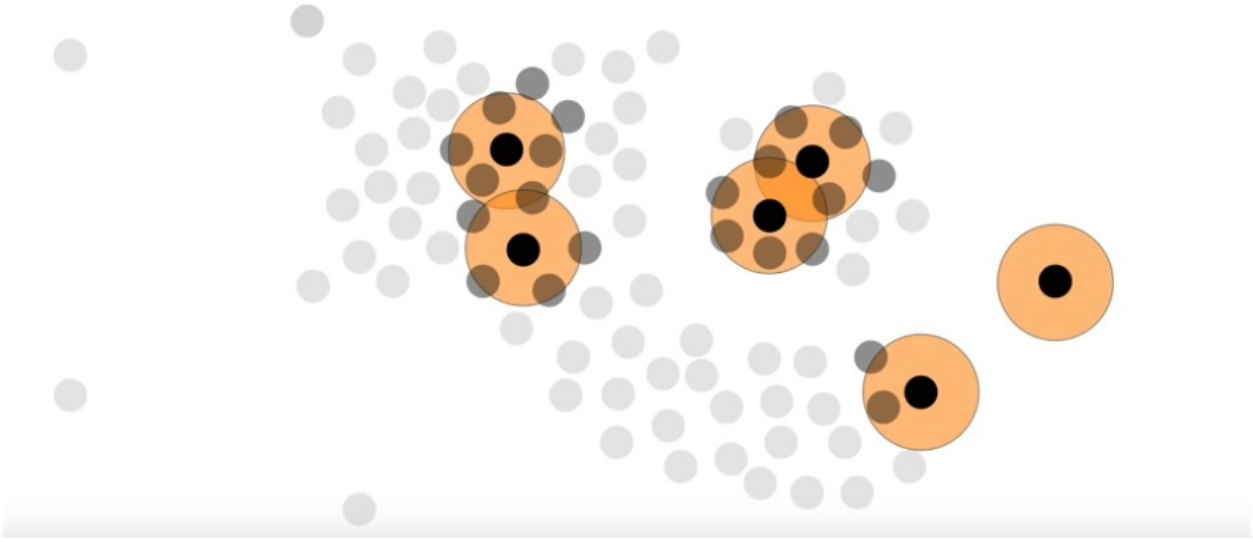



…is close to 5 other points because the orange circle overlaps, at least partially, 5 other points.

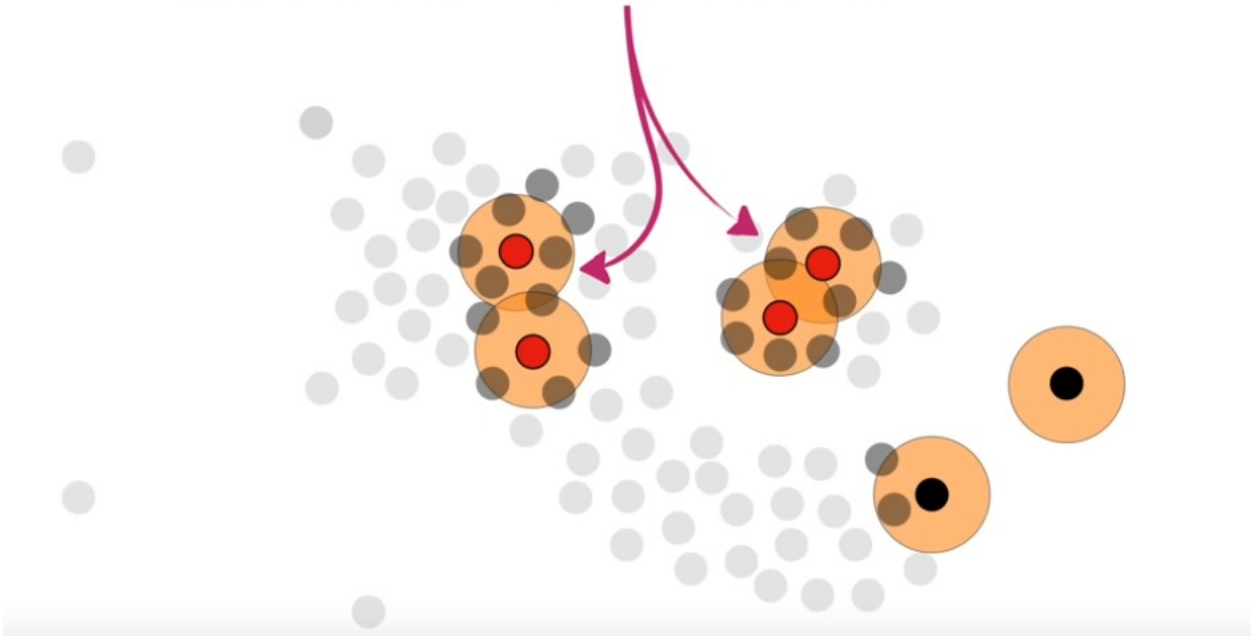

This red point is only close to 2 other points…


…and this red point is not close to any other point because the orange circle does not overlap anything else.

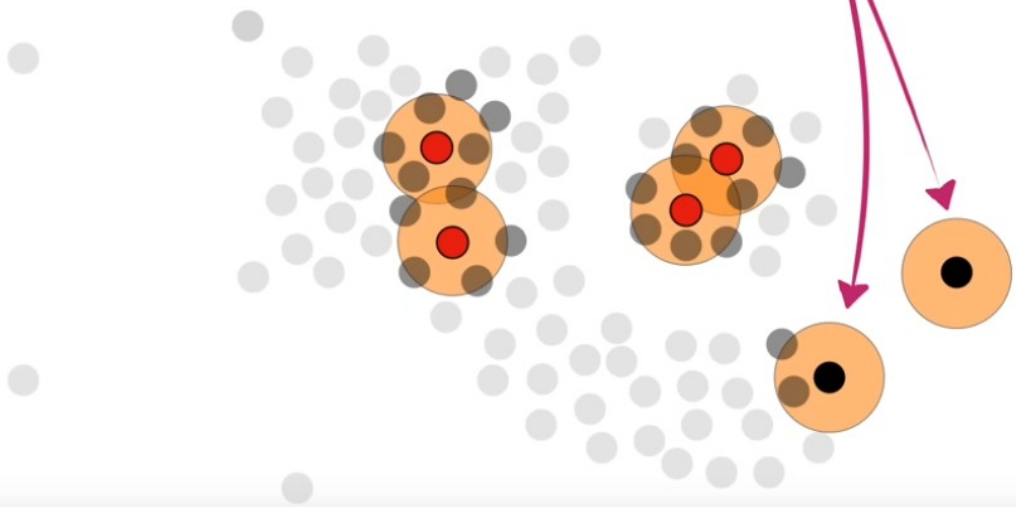
NOTE: The number of close points for a **Core Point** is user defined, so, when using **DBSCAN**, you might need to fiddle with this parameter as well.



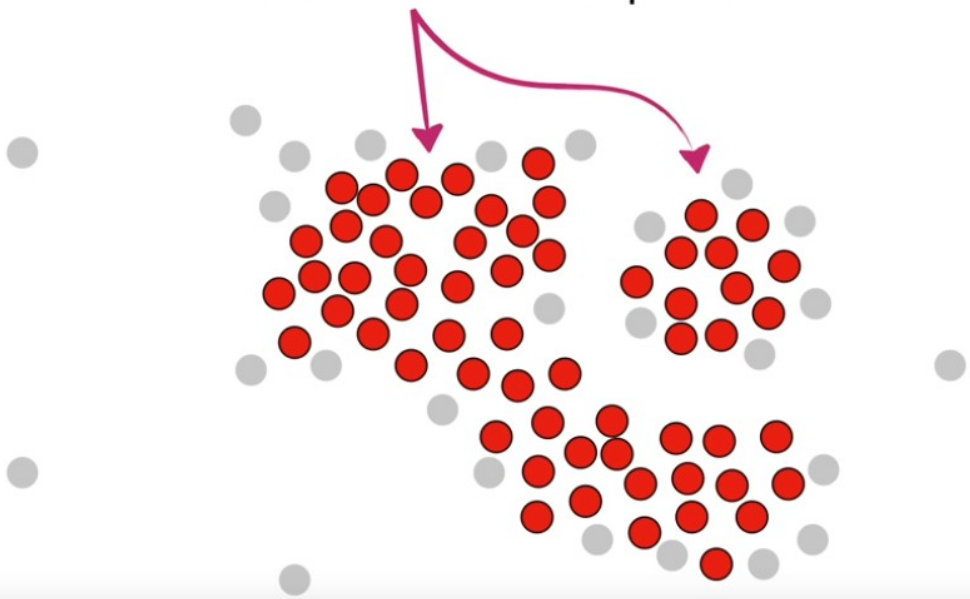
Anyway, these **4** points are some of the **Core Points**, because their **orange circles** overlap at least **4** other points…

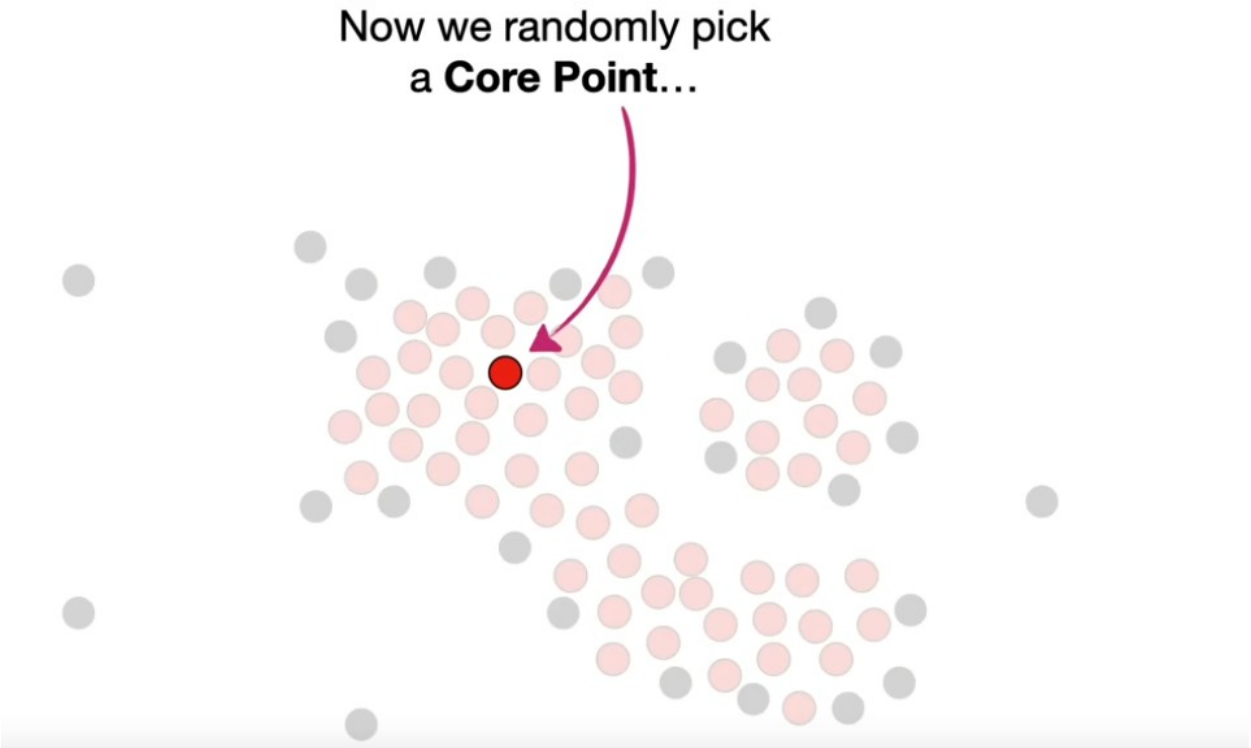


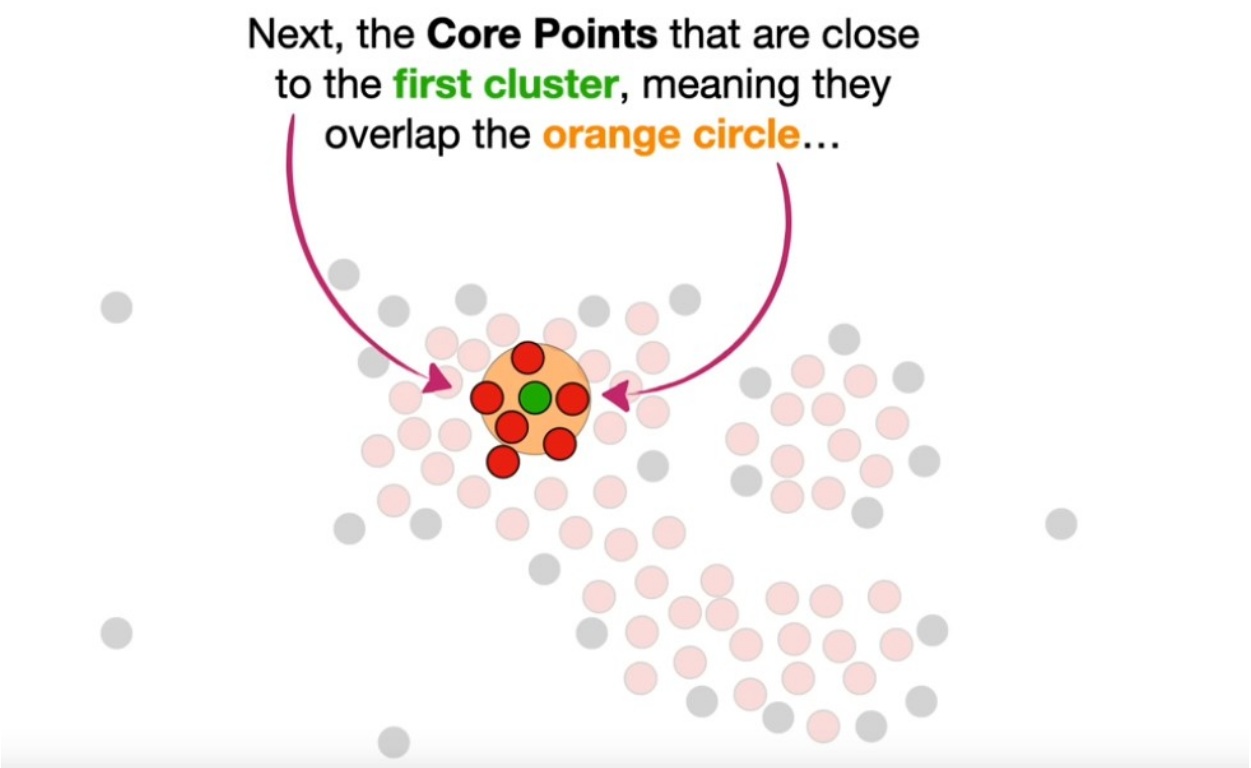
…but neither of these points are **Core Points** because their **orange circles** do not overlap **4** or more other points.



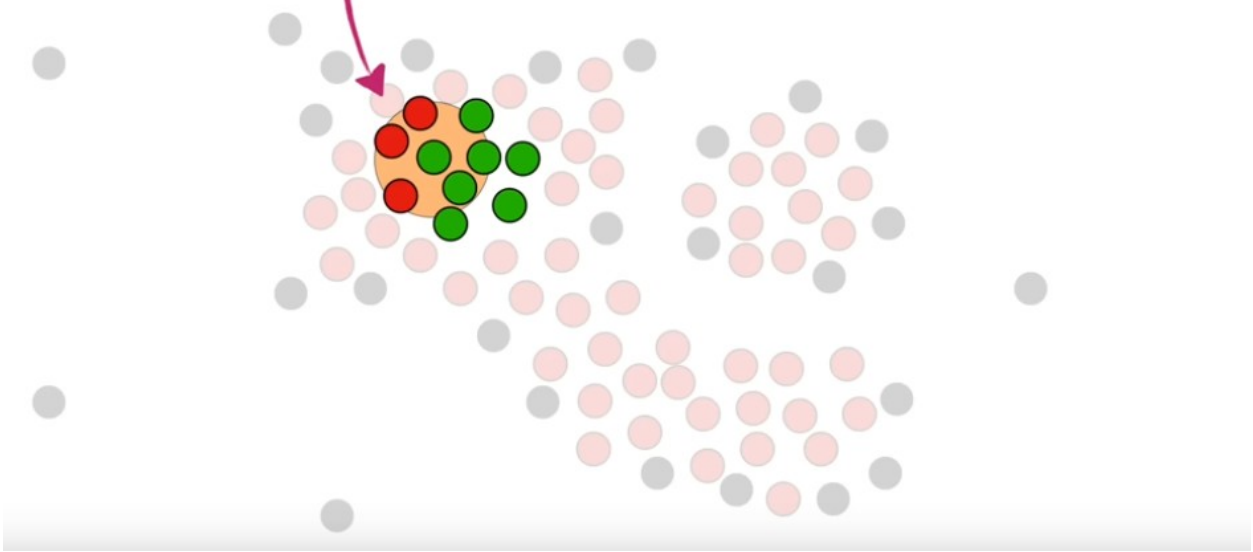
Ultimately, we can call all of these **red points** **Core Points** because they are all close to **4** or more other points…



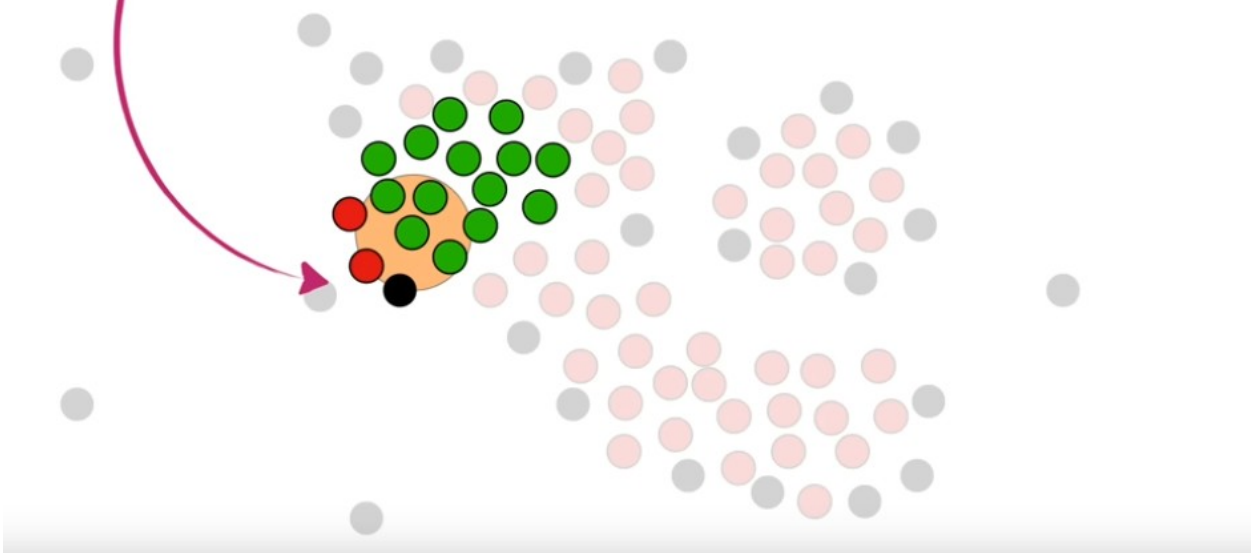

Now we randomly pick
a **Core Point**…


Next, the **Core Points** that are close
to the **first cluster**, meaning they
overlap the **orange circle**…

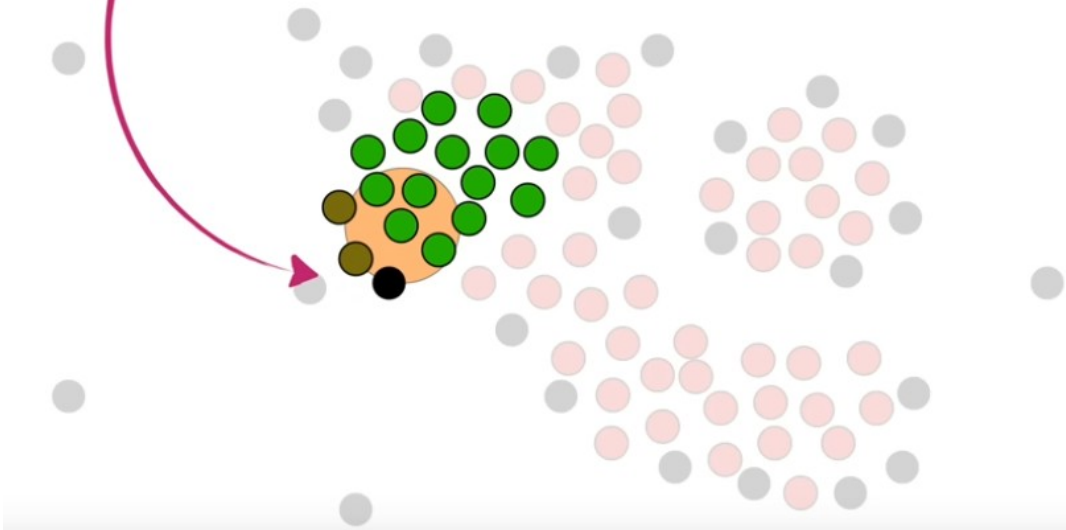
Then the **Core Points** that are close to the growing **first cluster** join it…



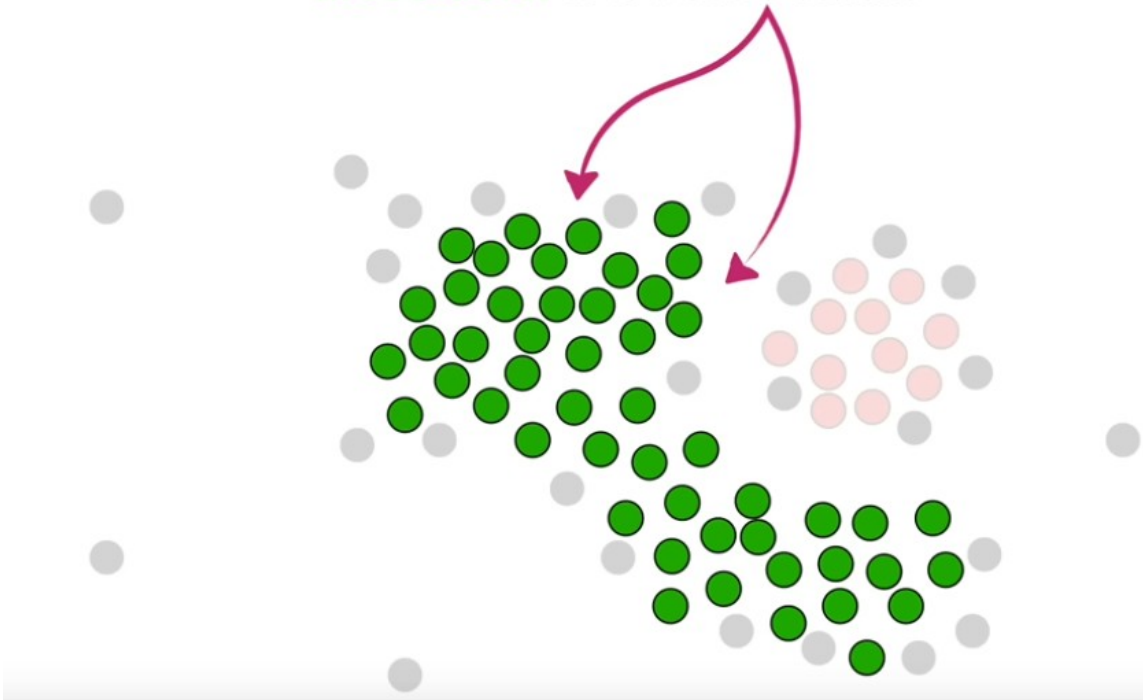
Here we see **2 Core Points** and **1 Non-Core Point** that are all close to the growing **first cluster**…



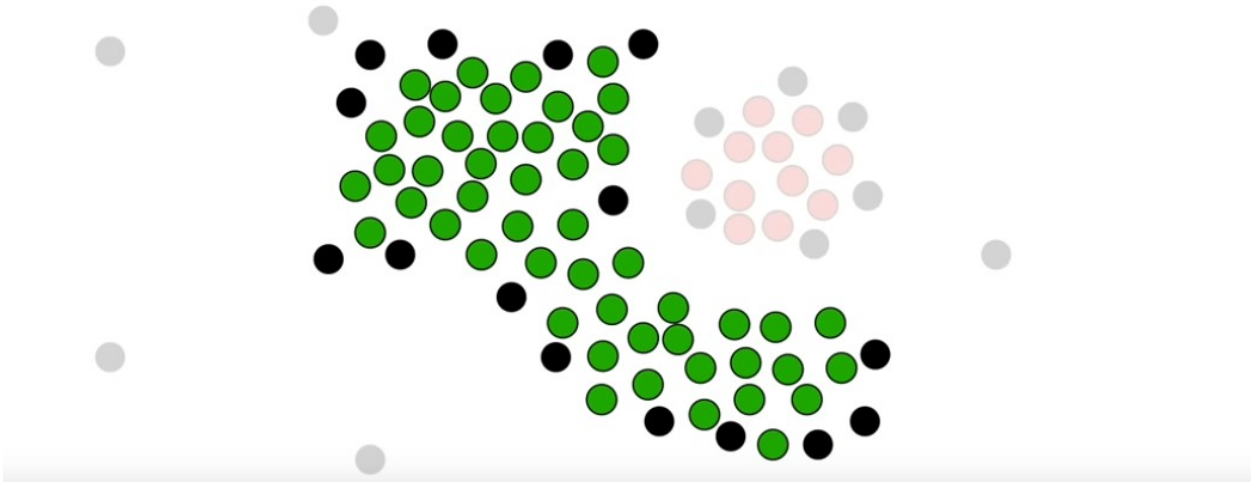
…and at this point, we only add the **Core Points** to the **first cluster**.



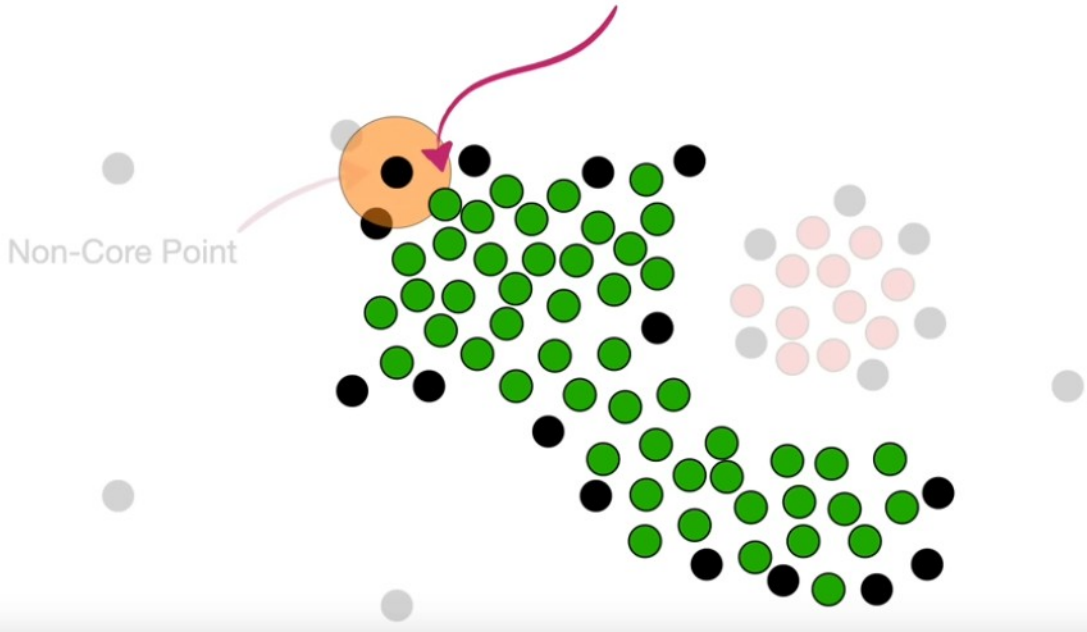
NOTE: At this point, every single point in the **first cluster** is a **Core Point**…

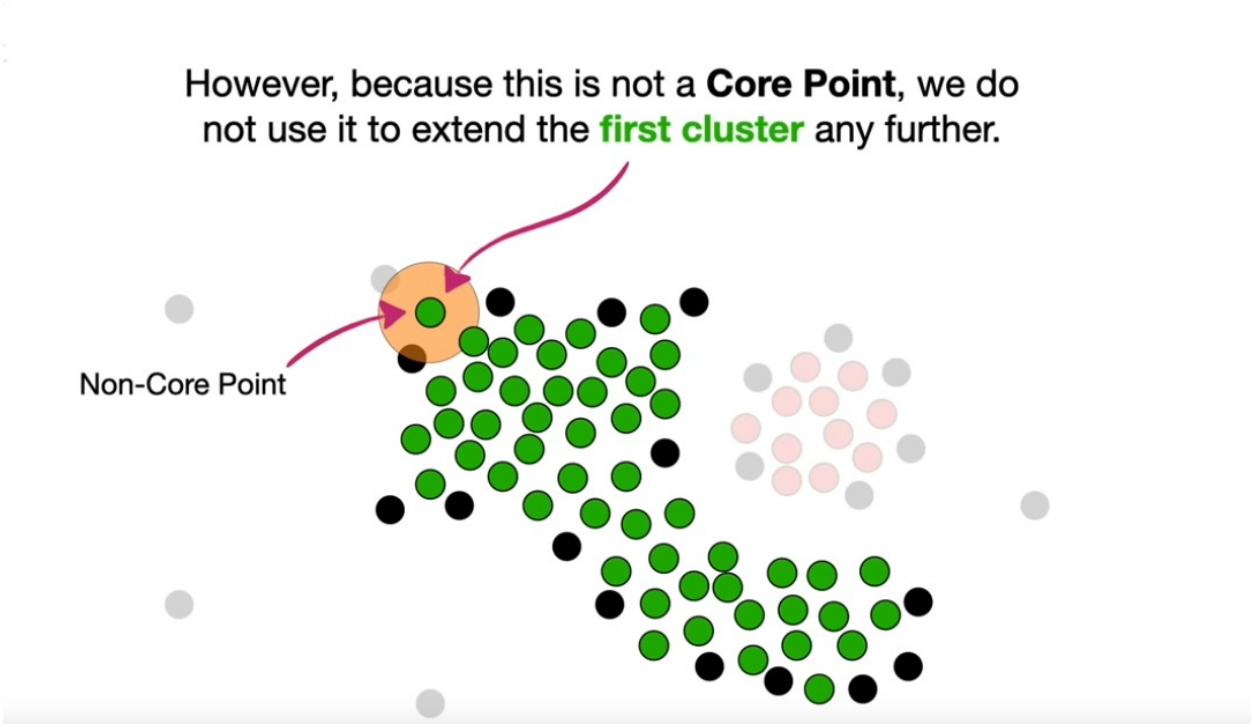


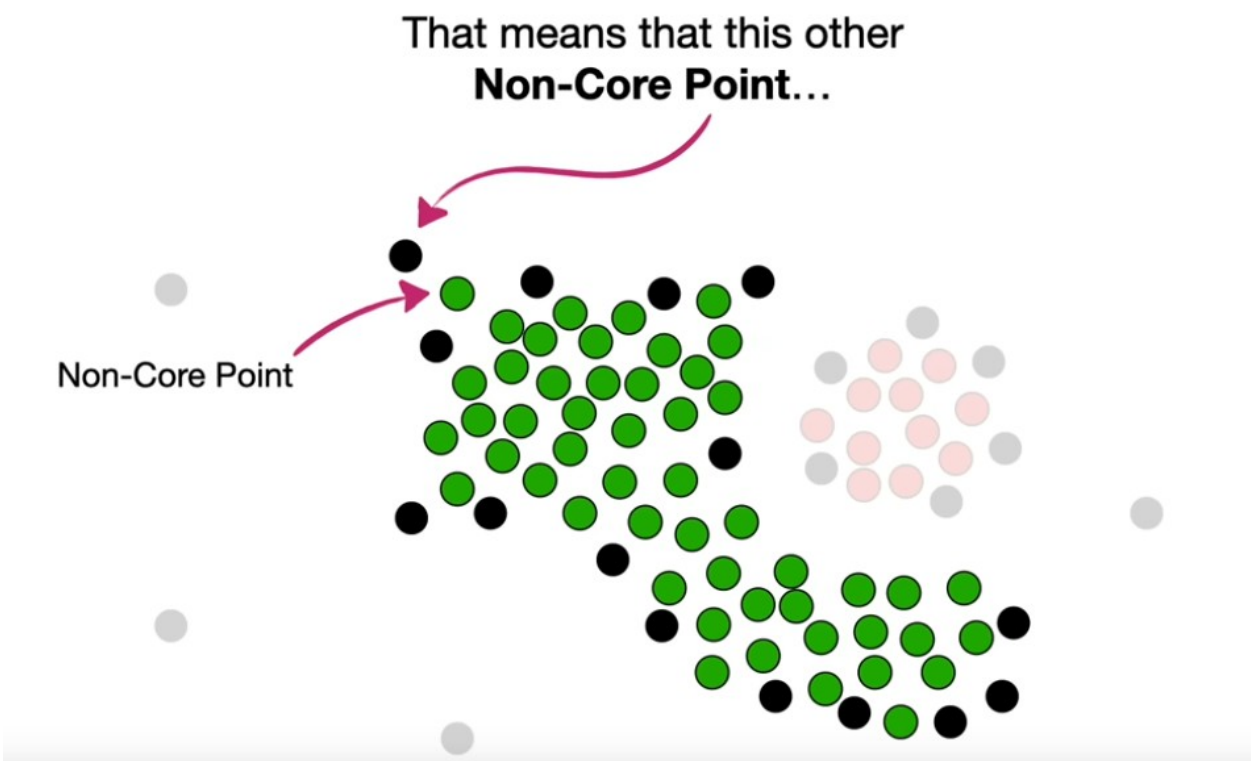
…we add all of the **Non-Core Points** that are close to **Core Points** in the **first cluster**.



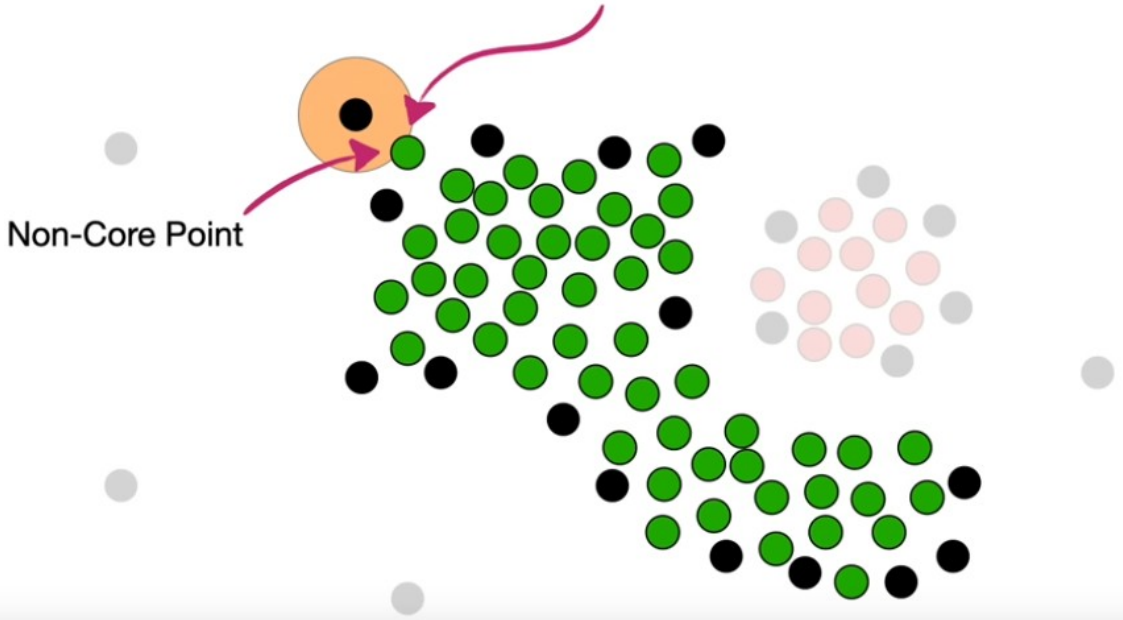
…is close to a **Core Point** in the **first cluster**…



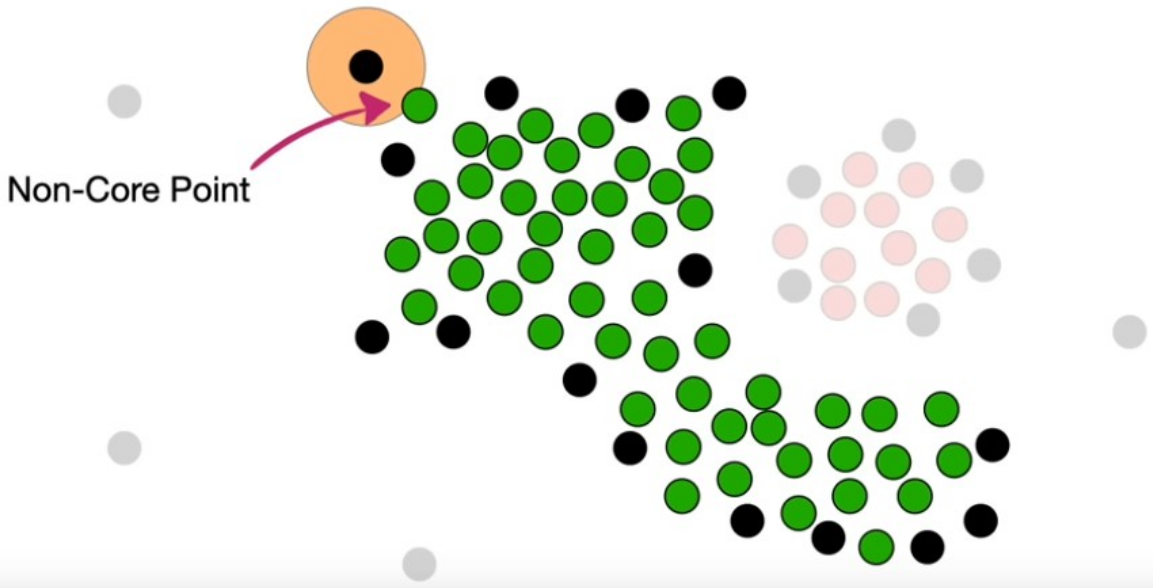

However, because this is not a **Core Point**, we do not use it to extend the **first cluster** any further.
Non-Core Point


That means that this other **Non-Core Point**…
Non-Core Point

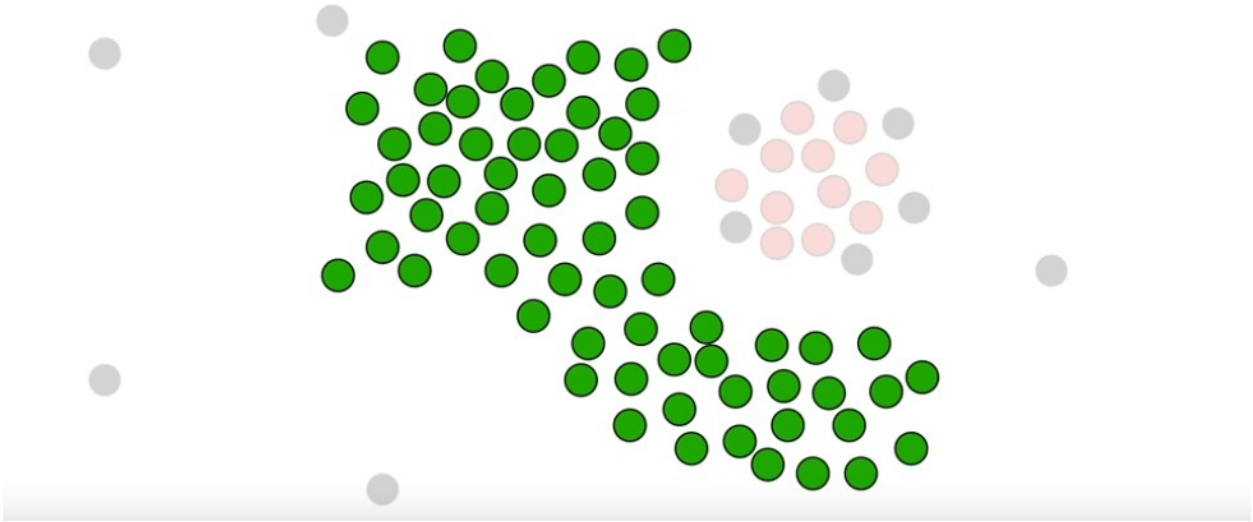
…which is close to the **Non-Core Point** that was just made part of the **first cluster**…



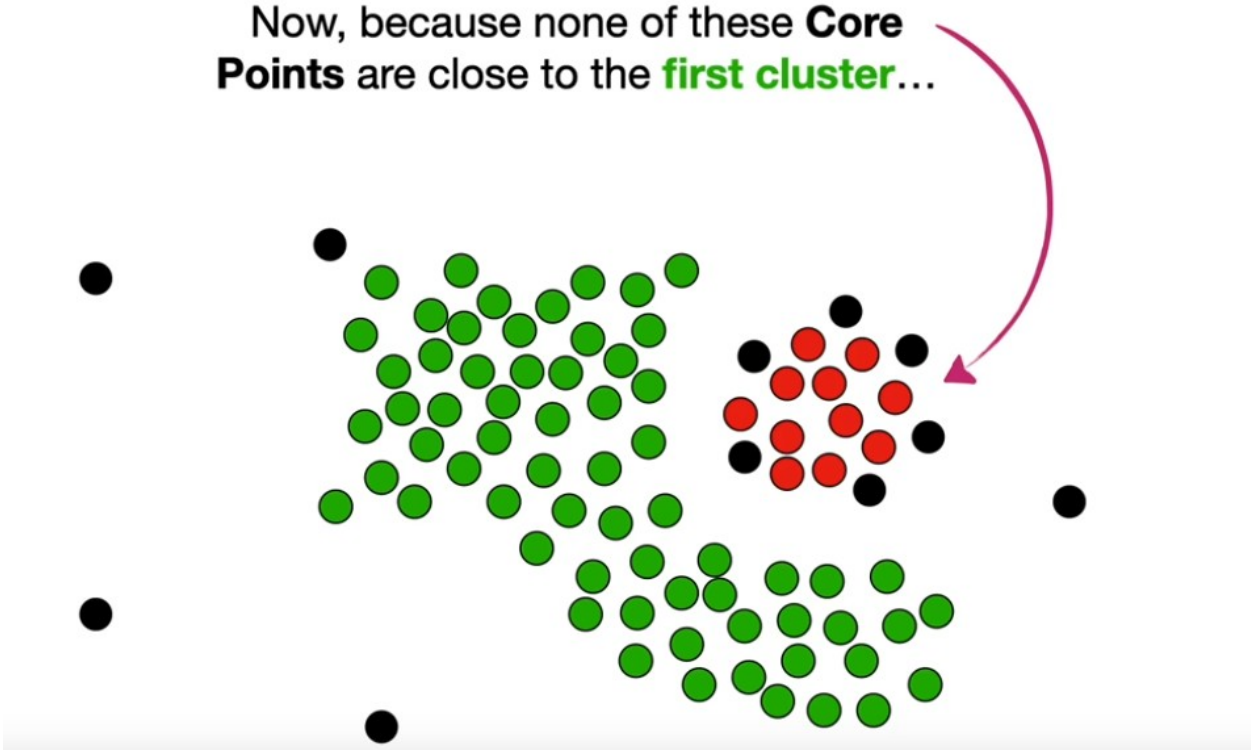
…will not be added to the **first cluster** because it is not close to a **Core Point**.



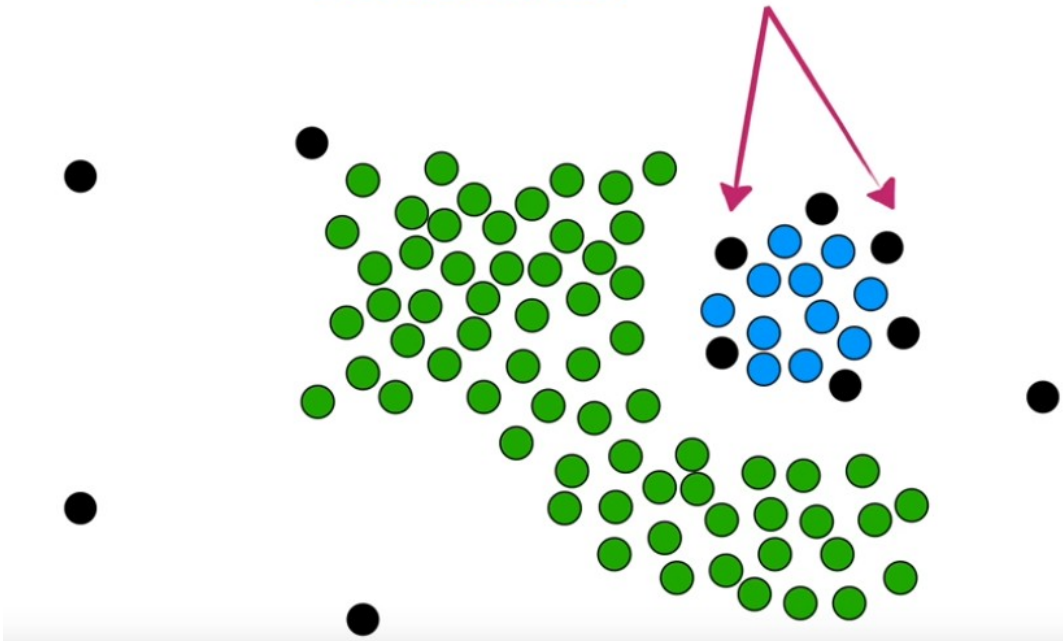
Now we add all of the **Non-Core Points** that are close **Core Points** in the **first cluster** to the **first cluster**.



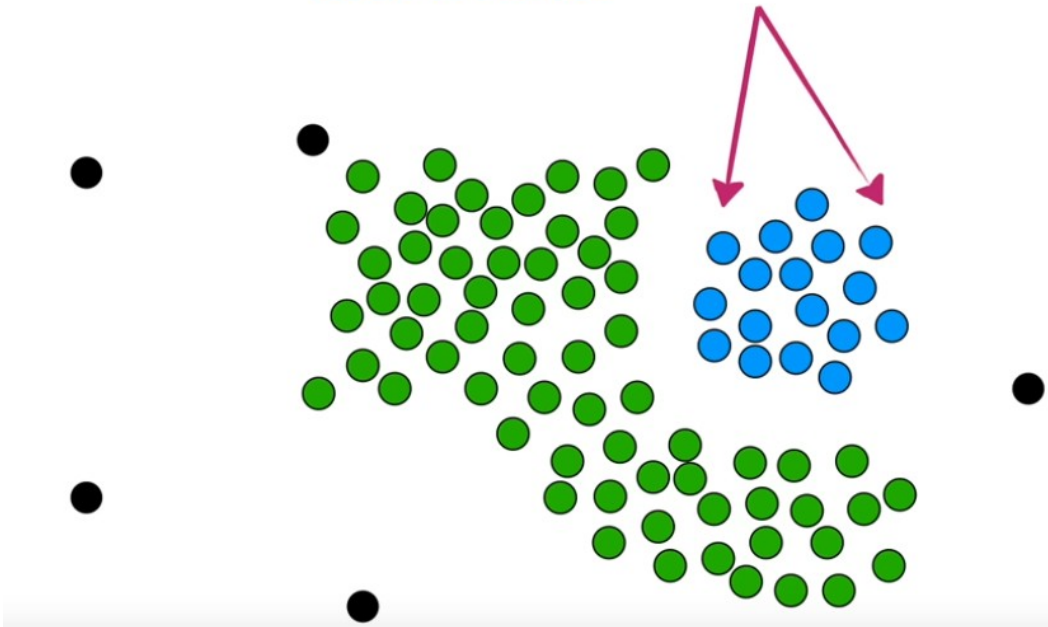
Now, because none of these **Core Points** are close to the **first cluster**…

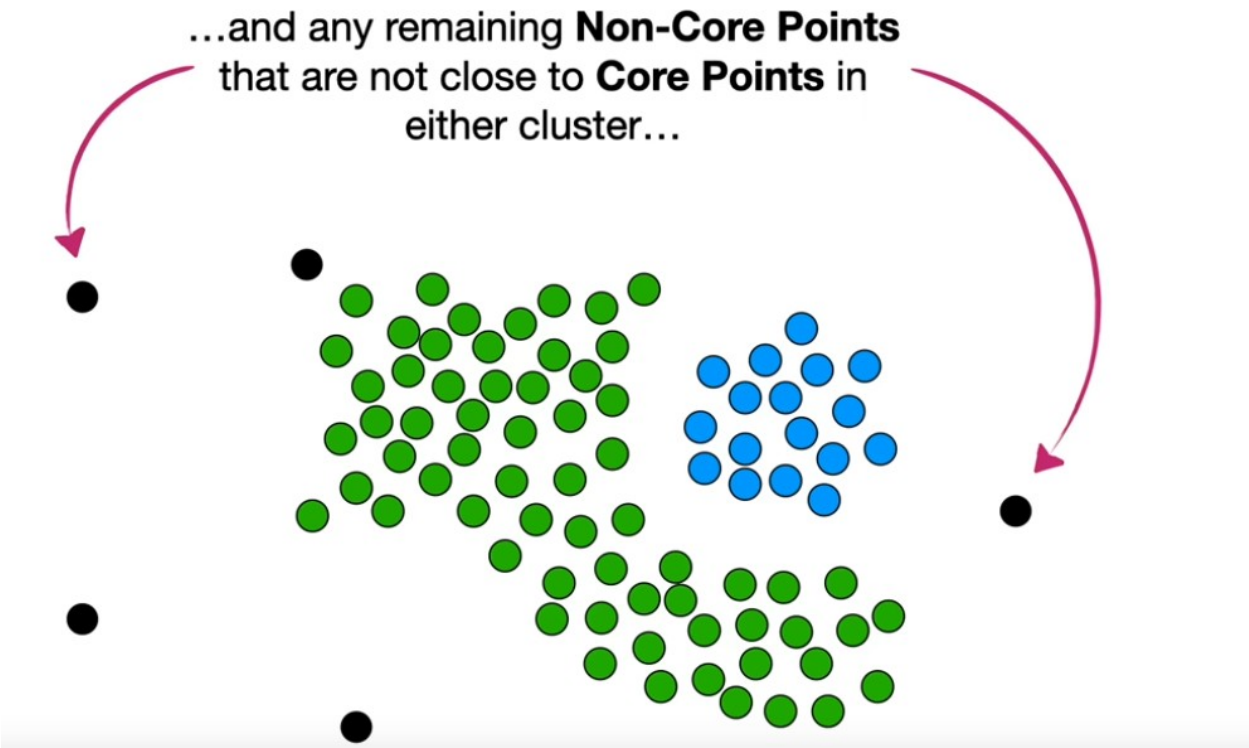


…and the **Non-Core Points** that are close to the **second cluster** are added to it.

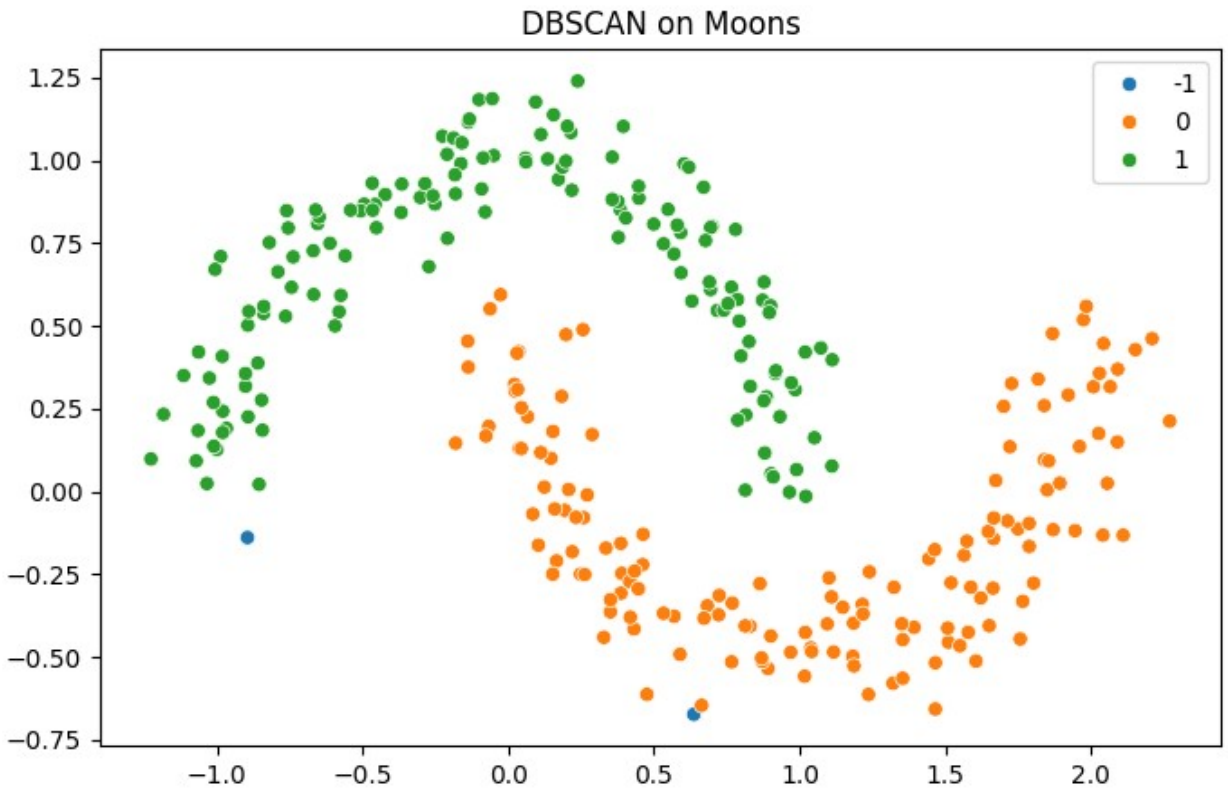


…and the **Non-Core Points** that are close to the **second cluster** are added to it.






```
# Fit DBSCAN
db = DBSCAN(eps=0.3, min_samples=5)
db_labels = db.fit_predict(Xm_scaled)

# Visualize DBSCAN result
plt.figure(figsize=(8, 5))
sns.scatterplot(x=X_moon[:, 0], y=X_moon[:, 1], hue=db_labels,
palette='tab10')
plt.title('DBSCAN on Moons')
plt.show()
```



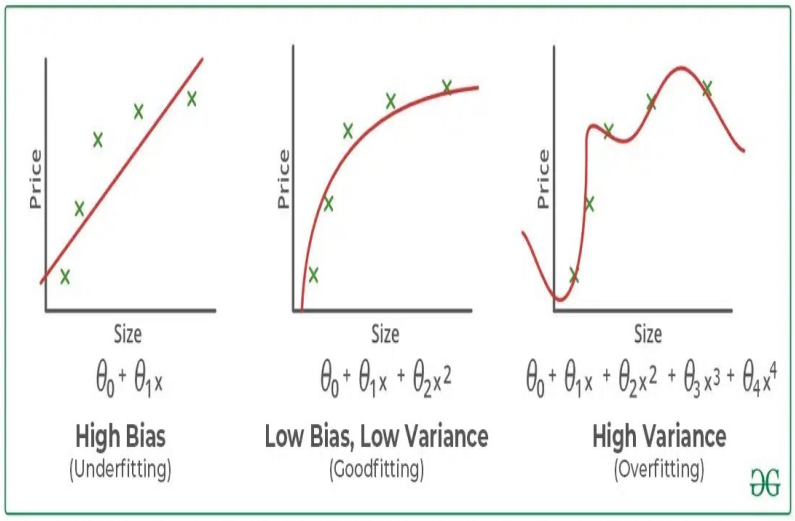

```
# 📊 Summary Table
from IPython.display import display, Markdown

display(Markdown("""
| Algorithm              | Needs K? | Handles Outliers? | Handles Non-
Spherical? |
|------------------------|----------|--------------------|------------
-------------|
| K-Means                | ✅ Yes   | ❌ No              | ❌ No
|
| Hierarchical Clustering| ✅ Yes   | ❌ No              | ❌ No
|
| DBSCAN                 | ❌ No    | ✅ Yes             | ✅ Yes
|

- Use **Hierarchical** when you want interpretability and dendrograms
- Use **DBSCAN** when you want to detect noise or clusters of
arbitrary shape
"""))
```

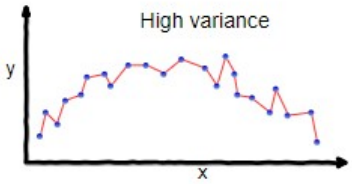
```
<IPython.core.display.Markdown object>
```

```
Train error =70 Percent  High Bias ( Its Performing Bad in Training
Data)
```

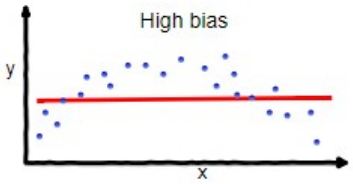
```
Train error = 1 percent  Low Bias ( Perfected Memorized Training Data)

test eror = 70   High Viarance (Perform very bad in Unseen Data )

Overfit -> Low Bias (Extremely Perfect Training Data ) + (High
Variance -> Unseen Data )

Low Bias (Extremely Perfect Training Data ) + (Low Variance -> Unseen
Data ) -> Perfect Model

High Bias ( Extremely Bad Model)
```

Underfit overfit Bias Variance

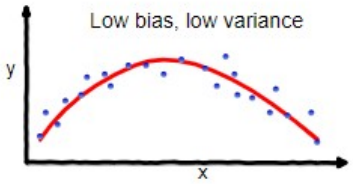




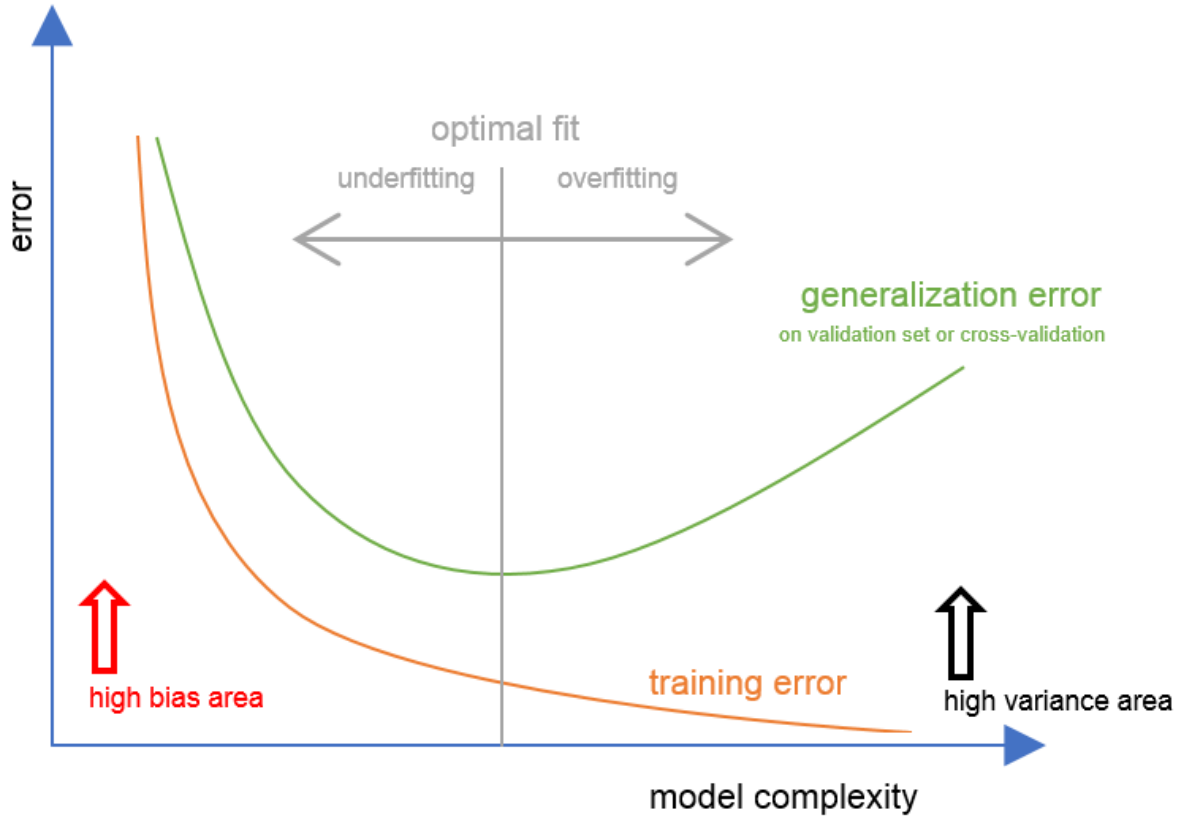
overfitting



underfitting



Good balance



 UNDERFITTING

Definition: The model is too simple to capture the underlying patterns in the data — it performs poorly on both training and test data. Symptoms:

```
High training error

High validation/test error

Model doesn’t improve much with training
```

 How to Fix:

```
Use a more complex model

    Upgrade from linear to polynomial model

    Try deeper neural networks or more sophisticated algorithms

Reduce regularization

    Lower the value of regularization parameters (e.g., decrease
lambda in Ridge/Lasso or C in SVM)
```

```
Train longer

    Increase number of training epochs (especially for neural
networks)

Feature engineering

    Add more relevant features

    Create interaction terms or nonlinear transformations

Lower bias algorithms

    Switch from simple models (like linear regression) to more
flexible ones (like random forests or gradient boosting)
```

 OVERFITTING

Definition: The model is too complex and learns the noise in the training data — performs well on training but poorly on test/validation data. Symptoms:

```
Low training error

High validation/test error

Performance drops on new/unseen data
```

 How to Fix:

```
Simplify the model

    Use fewer parameters, shallower trees, fewer layers

Add regularization

    Increase regularization strength (e.g., L1/L2 penalty)

    Use dropout (in neural nets)

More training data

    Overfitting often reduces with more examples

Early stopping

    Stop training when validation loss starts increasing

Data augmentation

    For image, audio, or text tasks, augment training data to improve
```

```
generalization

Cross-validation

    Use techniques like k-fold CV to tune hyperparameters robustly
```