

# MINERÍA DE DATOS

**Maximiliano Ojeda**

[muojeda@uc.cl](mailto:muojeda@uc.cl)

---



IIC-2433

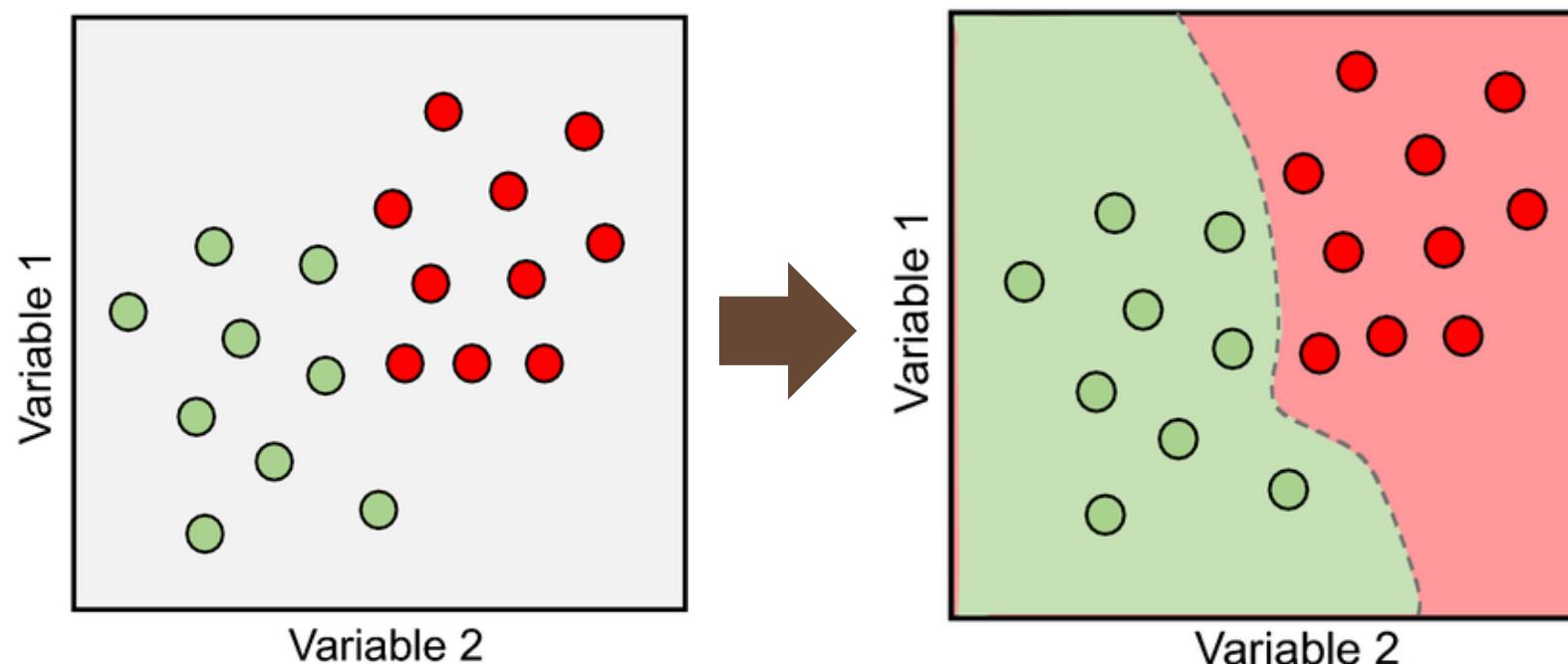


# Aprendizaje No Supervisado

# Tipos de Aprendizaje

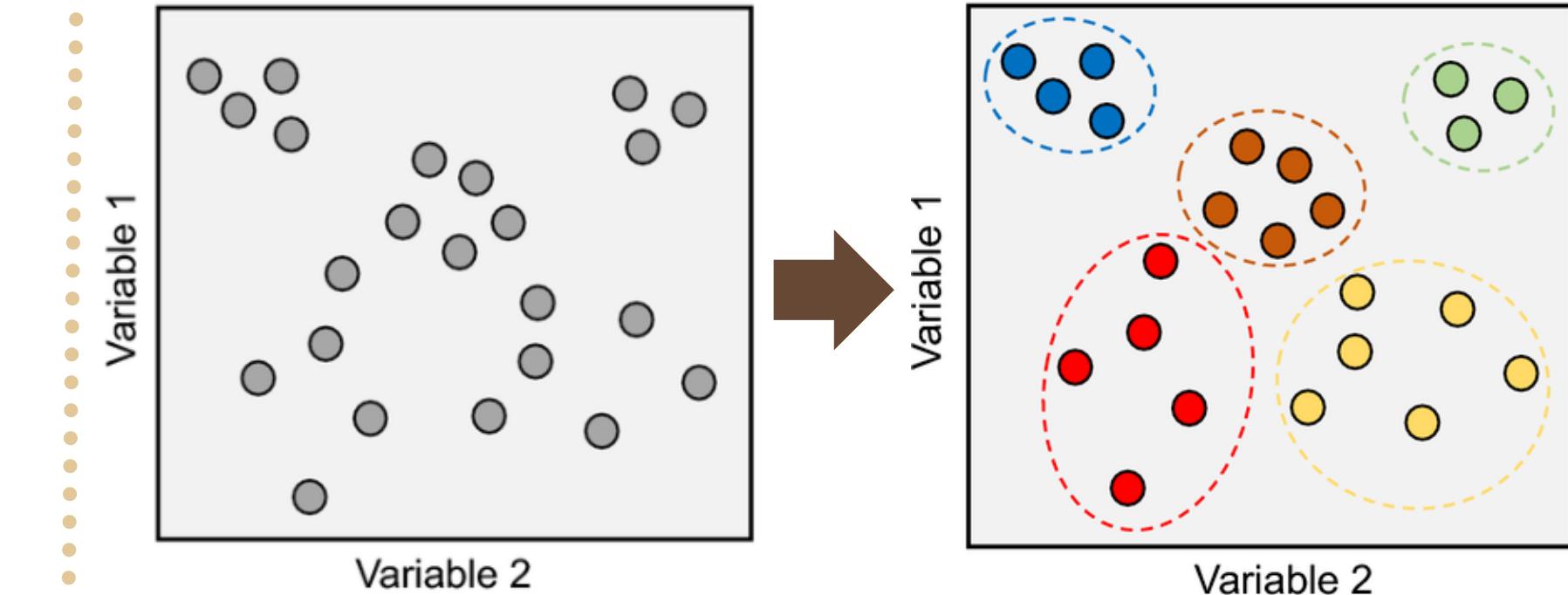
## Aprendizaje Supervisado

Los datos están etiquetados (existen un X y un Y).  
Los modelos aprenden a predecir la etiqueta Y



## Aprendizaje No Supervisado

Sólo tenemos X, Los modelos buscan patrones ocultos o estructuras



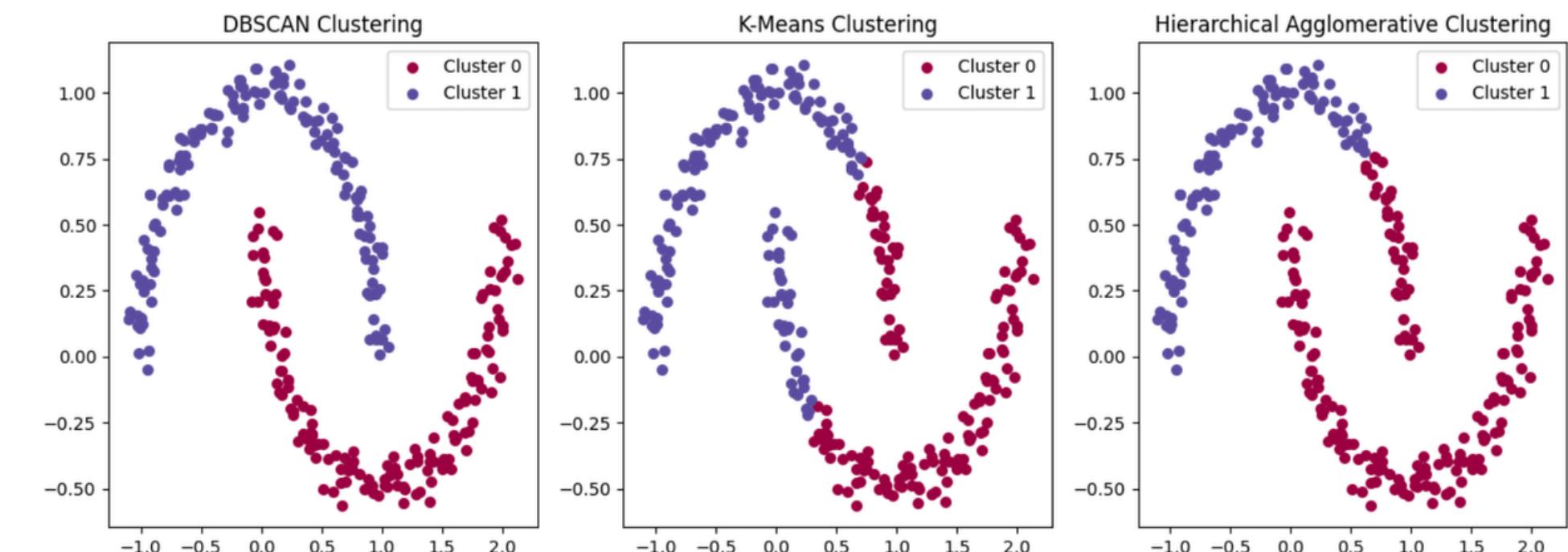
# Clustering

Clustering es una técnica de aprendizaje no supervisado cuyo objetivo es agrupar un conjunto de datos en grupos (clusters) de manera que:

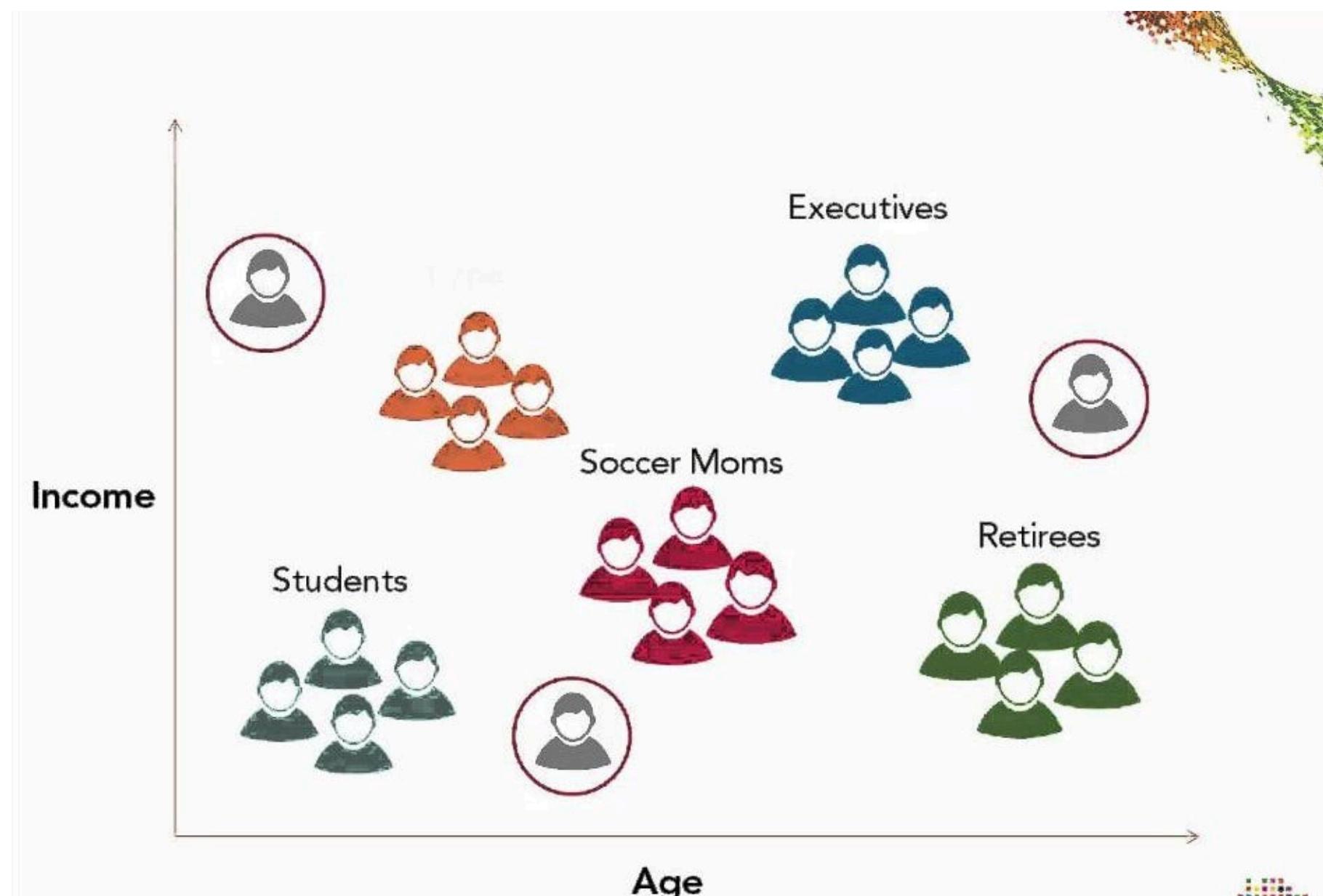
- Los elementos dentro de un mismo grupo sean lo más similares entre sí.
- Los elementos de grupos diferentes sean lo más distintos posible.

## Aplicaciones

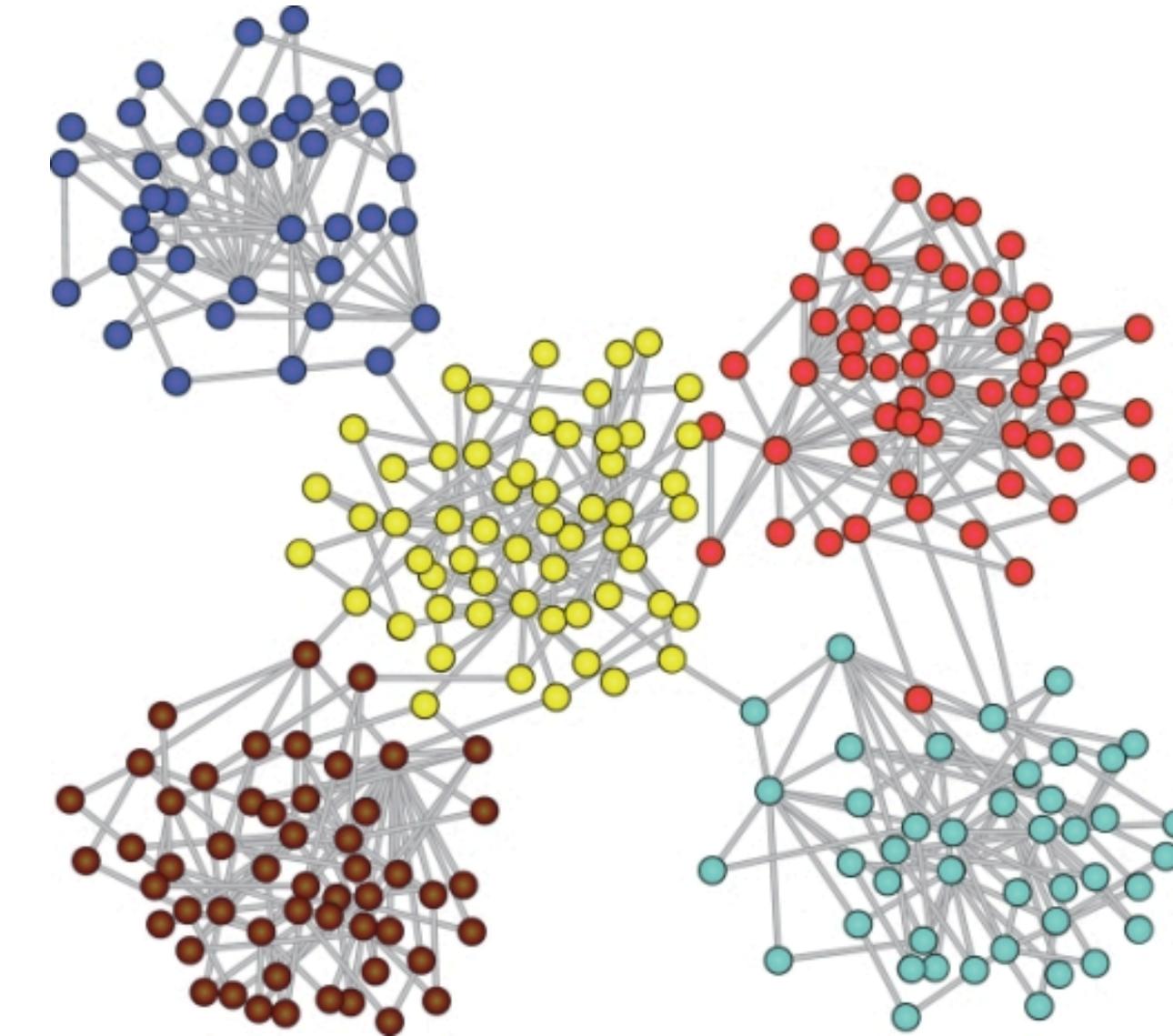
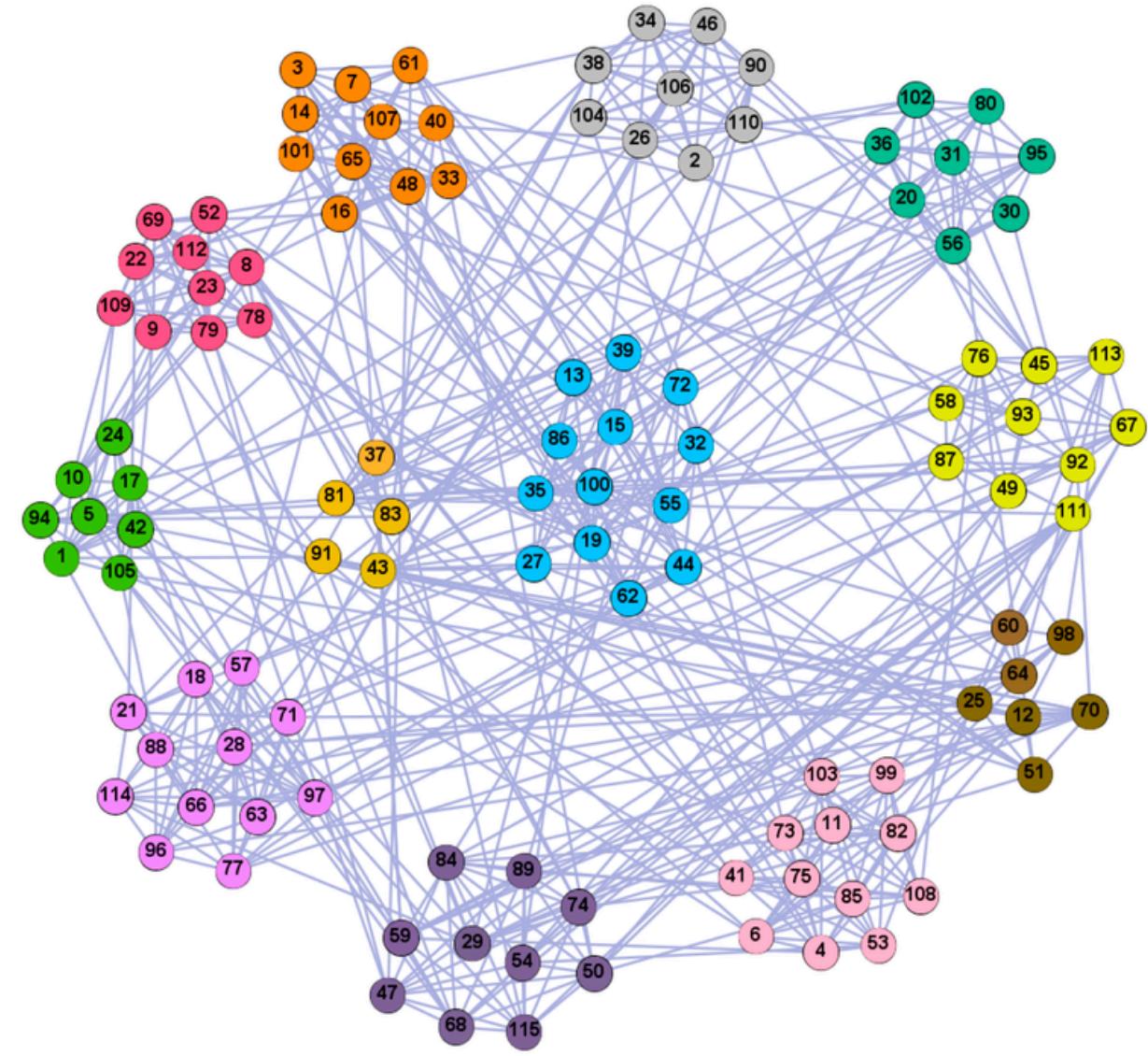
- Segmentación de clientes
- Análisis de imágenes
- Detección de anomalías
- Agrupación de documentos/textos.
- Biología y medicina.



# Clustering



# Clustering



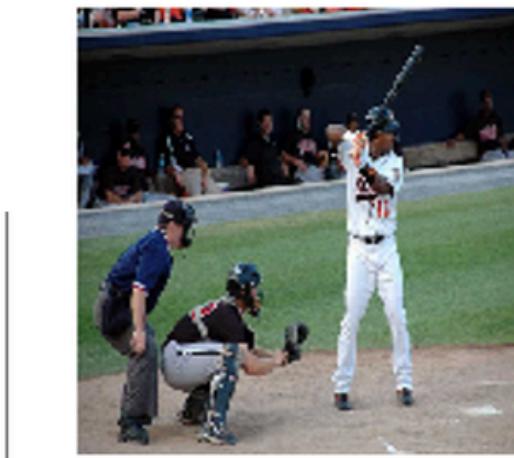
# Clustering



Segmentation



Segmentation



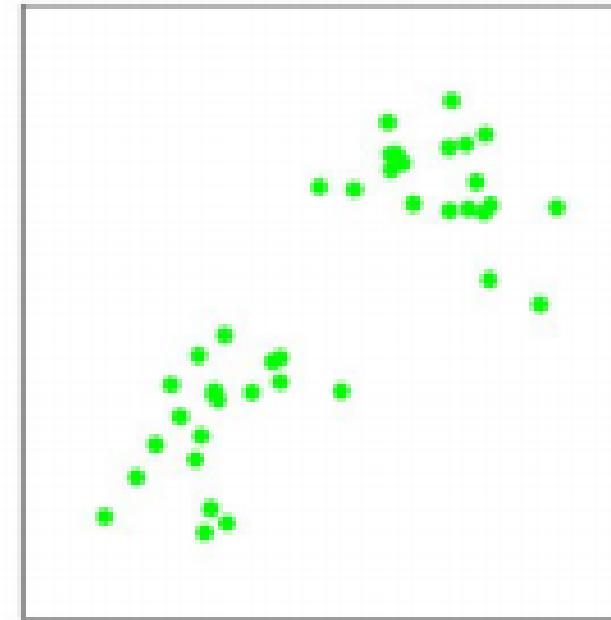
Segmentation



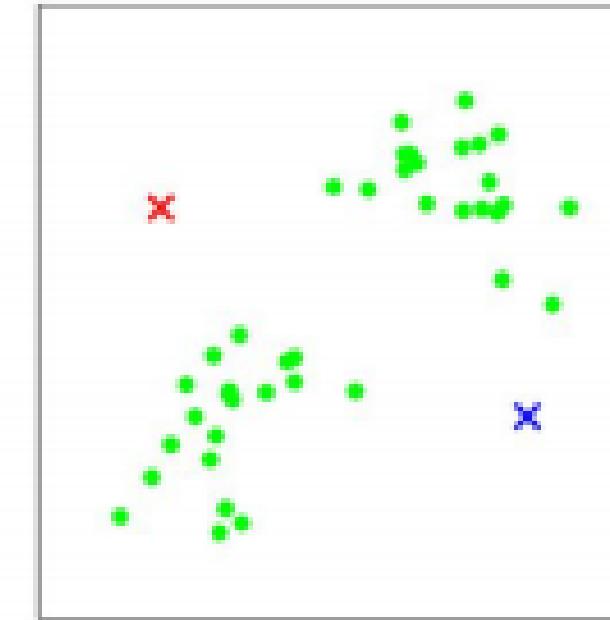


# K-Means

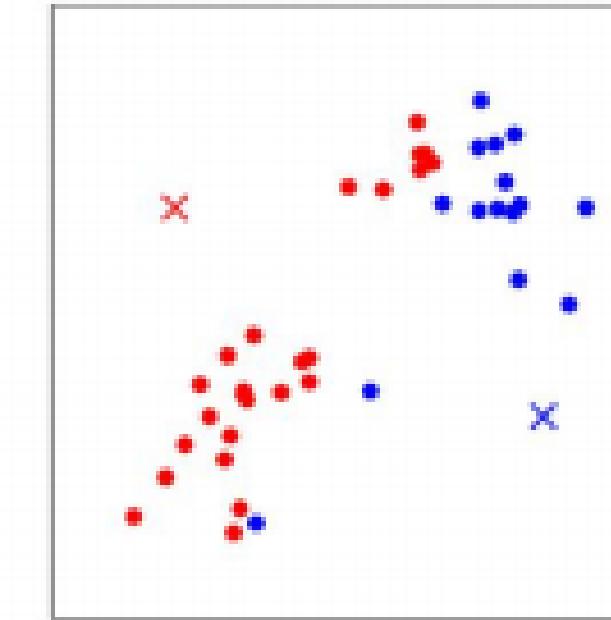
# K-Means



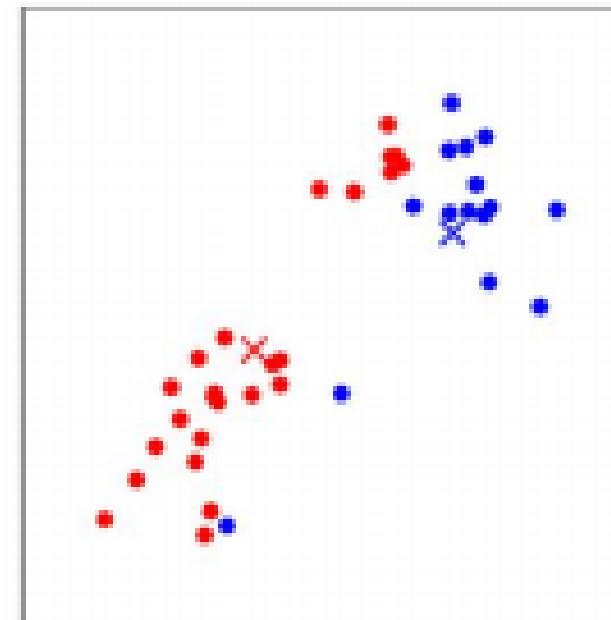
(a)



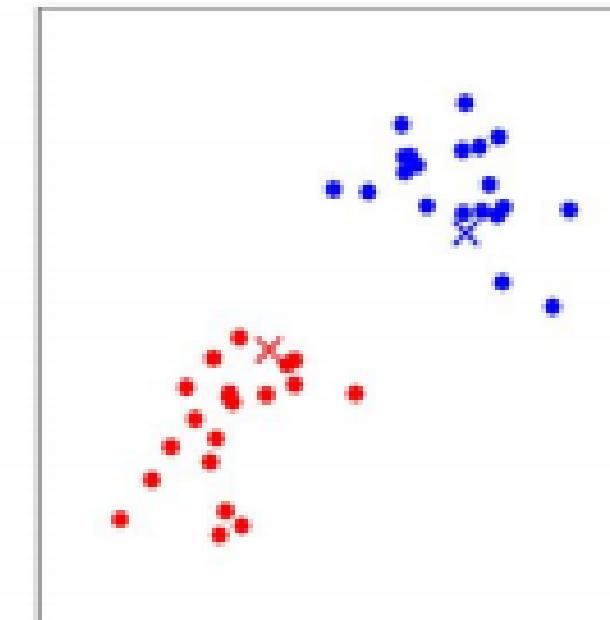
(b)



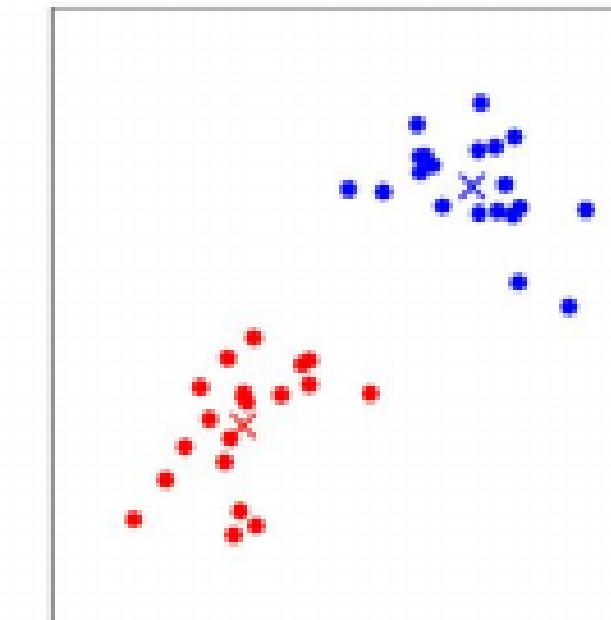
(c)



(d)



(e)

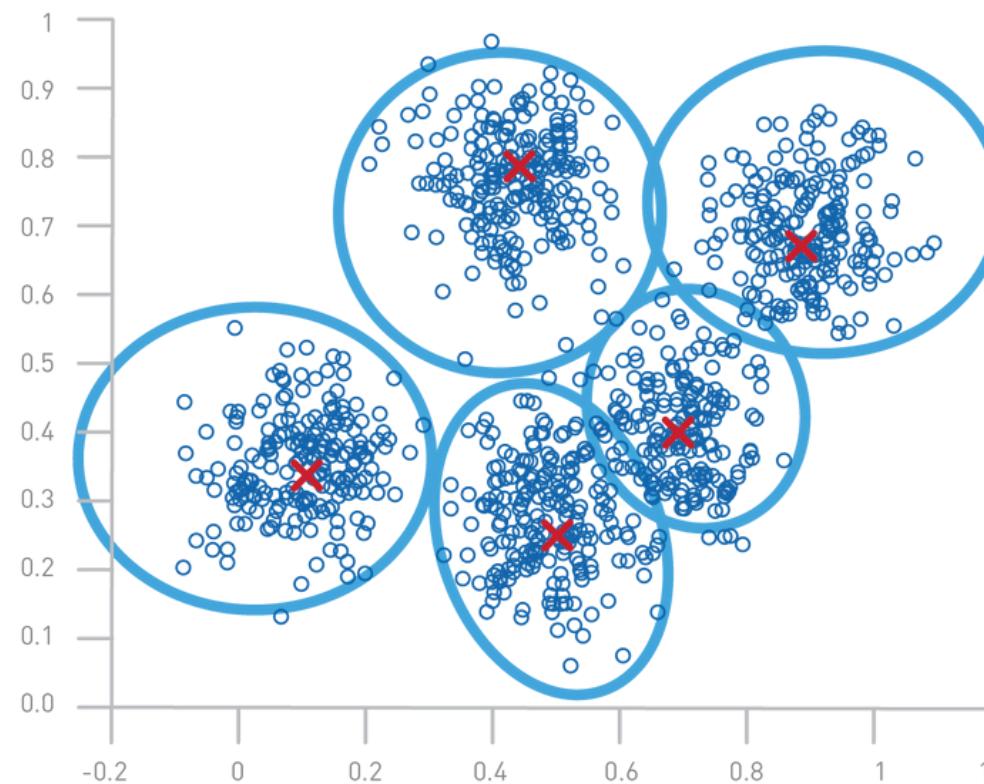


(f)

# K-Means

K-Means funciona como un algoritmo iterativo que divide los datos en k grupos (clusters) según su cercanía:

1. **Elegir k:** decides cuántos clusters quieres.
2. **Iniciar centroides:** se colocan **k** puntos al azar (los “centros” de cada cluster).
3. **Asignar puntos:** cada dato se asigna al **centroide más cercano** (ejemplo, distancia euclídea).
4. **Actualizar centroides:** calcular el promedio de los puntos en cada cluster y se mueve el centroide a esa posición.
5. **Repetir** pasos 3–4 hasta que los centroides ya no cambien (o número máximo de iteraciones).



# K-Means

## Problema

Tenemos el conjunto de datos:

$$X = \{x_1, x_2, \dots, x_n\}, \quad x_i \in \mathbb{R}^d$$

El objetivo es dividirlos en **k** clusters

$$C = \{C_1, C_2, \dots, C_k\}.$$

## Función Objetivo

Minimizar las distancias entre los puntos y el centroide de su cluster:

$$J = \sum_{j=1}^k \sum_{x_i \in C_j} \|x_i - \mu_j\|^2$$

## Algoritmo iterativo

**Inicialización:** elige aleatoriamente **k** centroides  $\mu_1, \dots, \mu_k$

**Asignación:** cada punto se asigna al cluster más cercano:

$$C_j = \{x_i : \|x_i - \mu_j\|^2 \leq \|x_i - \mu_l\|^2 \quad \forall l\}$$

**Actualización:** recalcular el centroide de cada cluster:

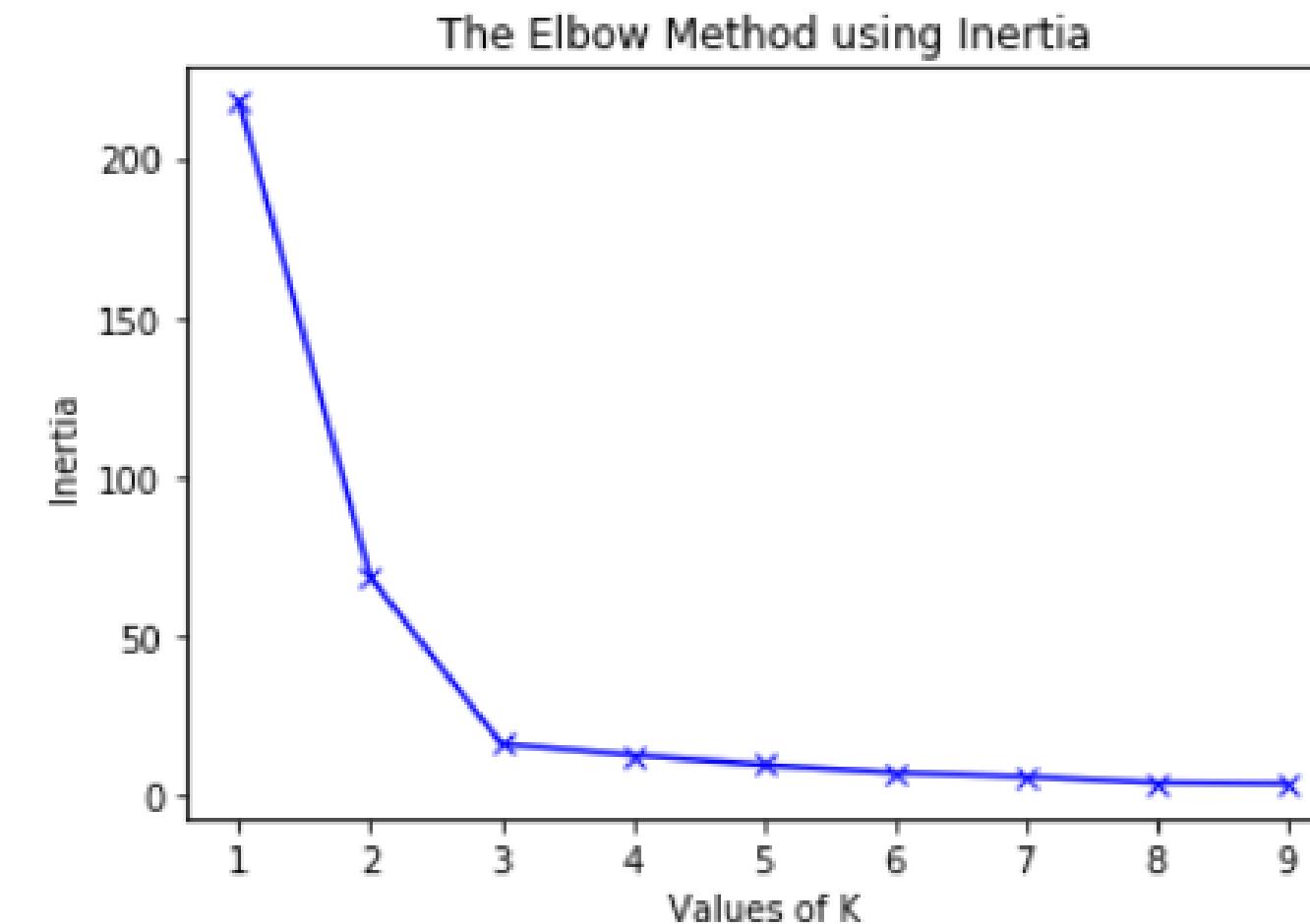
$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

Repetir hasta que los centroides no cambien.

# Método del codo (Elbow)

- Calcular la función objetivo (también llamada **inertia**) para distintos valores de **k**.
- A medida que aumenta **k**, la inercia disminuye (más clusters = menor error).
- Se elige el **k** en el que la mejora empieza a ser marginal → el “codo” de la curva.

$$\text{Inertia}(k) = \sum_{j=1}^k \sum_{x_i \in C_j} \|x_i - \mu_j\|^2$$



# K-Means

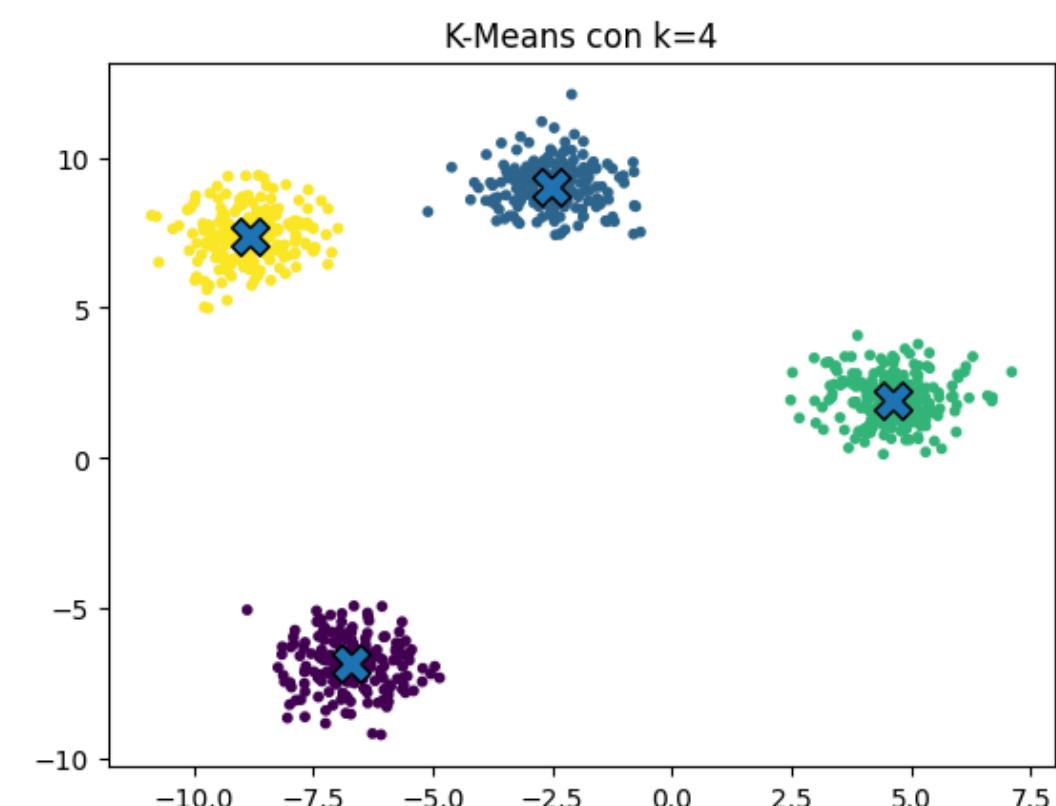
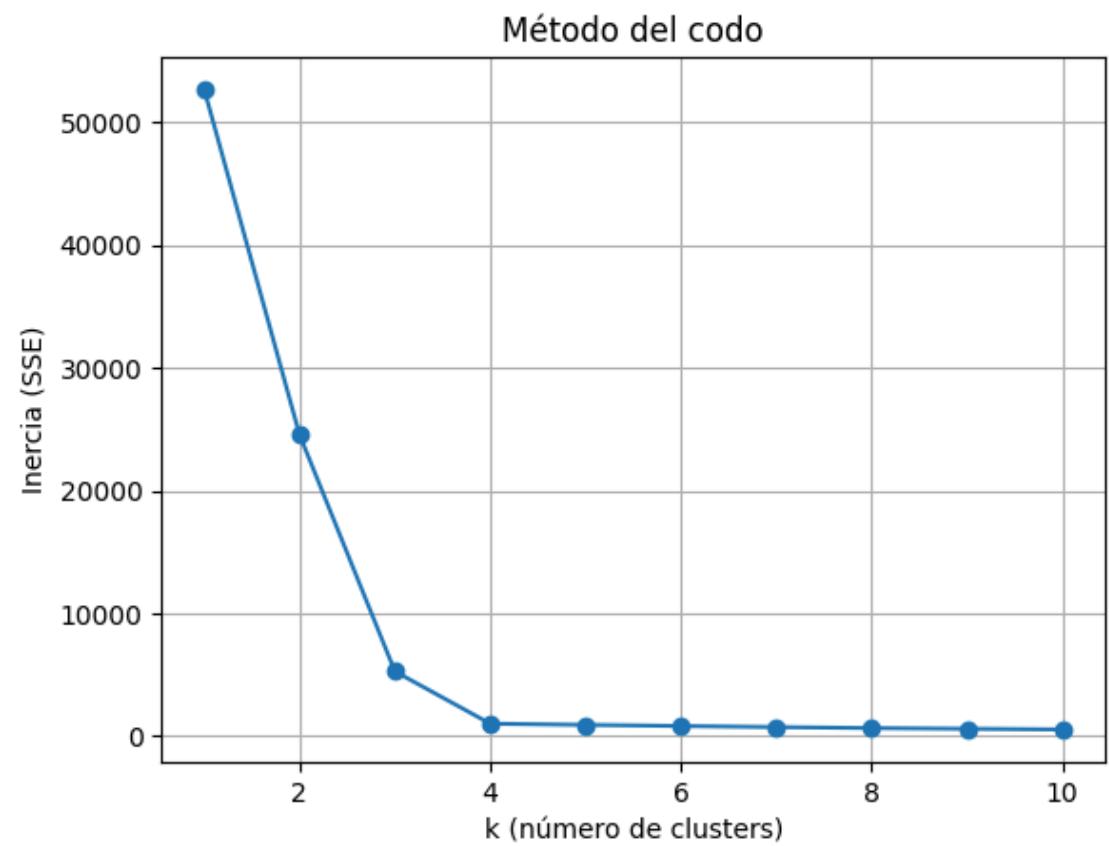
```
from sklearn.cluster import KMeans

X, y_true = make_blobs(n_samples=800, centers=4, cluster_std=0.80)

# 2) Cálculo de inercia para distintos k
Ks = range(1, 11)
inertias = []
for k in Ks:
    km = KMeans(n_clusters=k, n_init=10, random_state=42)
    km.fit(X)
    inertias.append(km.inertia_)

# Eligir k mirando el “codo” de la curva
k_opt = 4

# K-Means con k_opt
km_final = KMeans(n_clusters=k_opt, n_init=10, random_state=42)
labels = km_final.fit_predict(X)
centers = km_final.cluster_centers_
print("Inercia con k_opt:", km_final.inertia_)
print("Centroides:\n", centers)
```





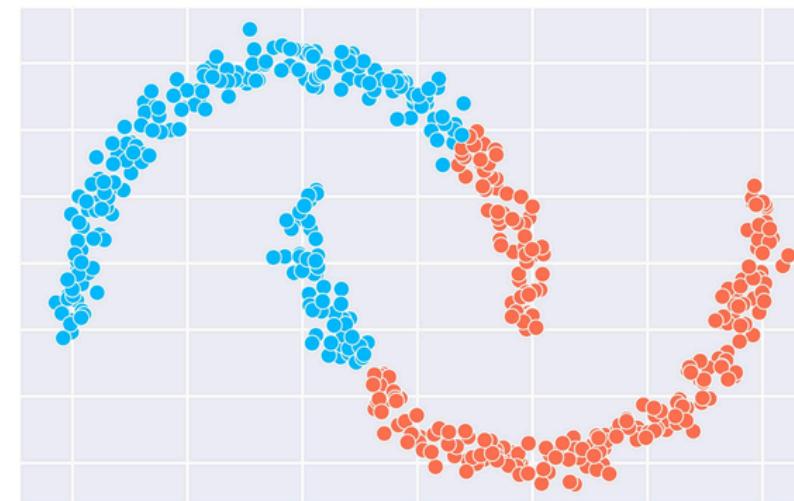
# DBSCAN

# ¿Por qué necesitamos otro algoritmo?

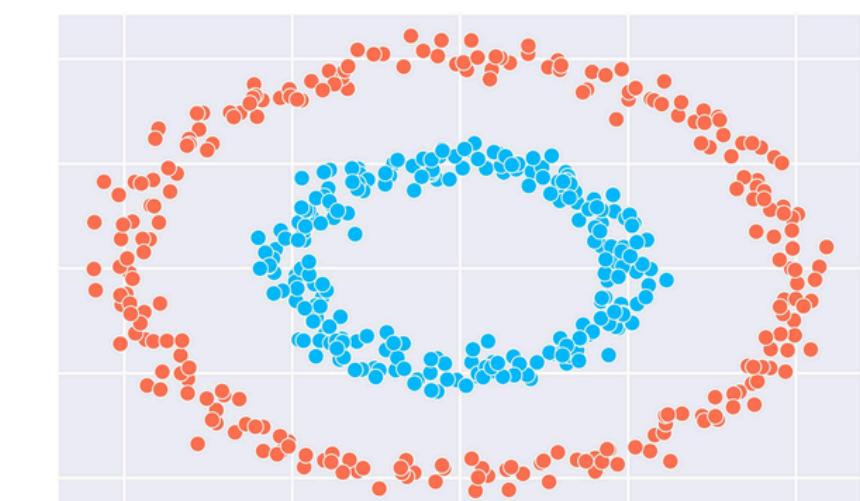
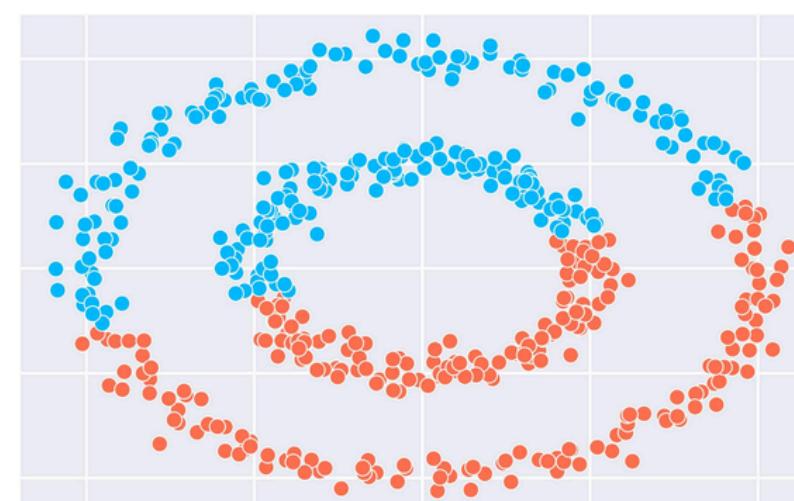
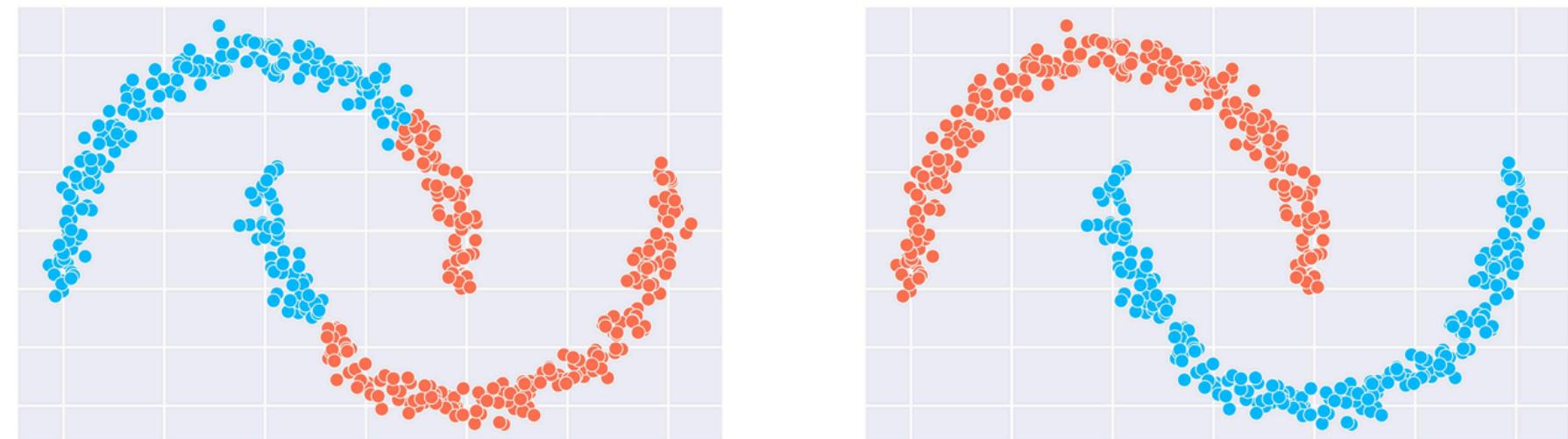
## Limitaciones de K-Means

- Asume que los clusters son aproximadamente esféricos y de tamaño similar.
- Es **sensible a outliers**, un solo dato muy alejado puede desplazar fuertemente un centroide.

KMeans



DBSCAN



# DBSCAN

**DBSCAN (Density-Based Spatial Clustering of Applications with Noise)** funciona de manera distinta a K-Means: en lugar de depender de centroides y un número fijo de clusters, agrupa puntos según su densidad en el espacio.

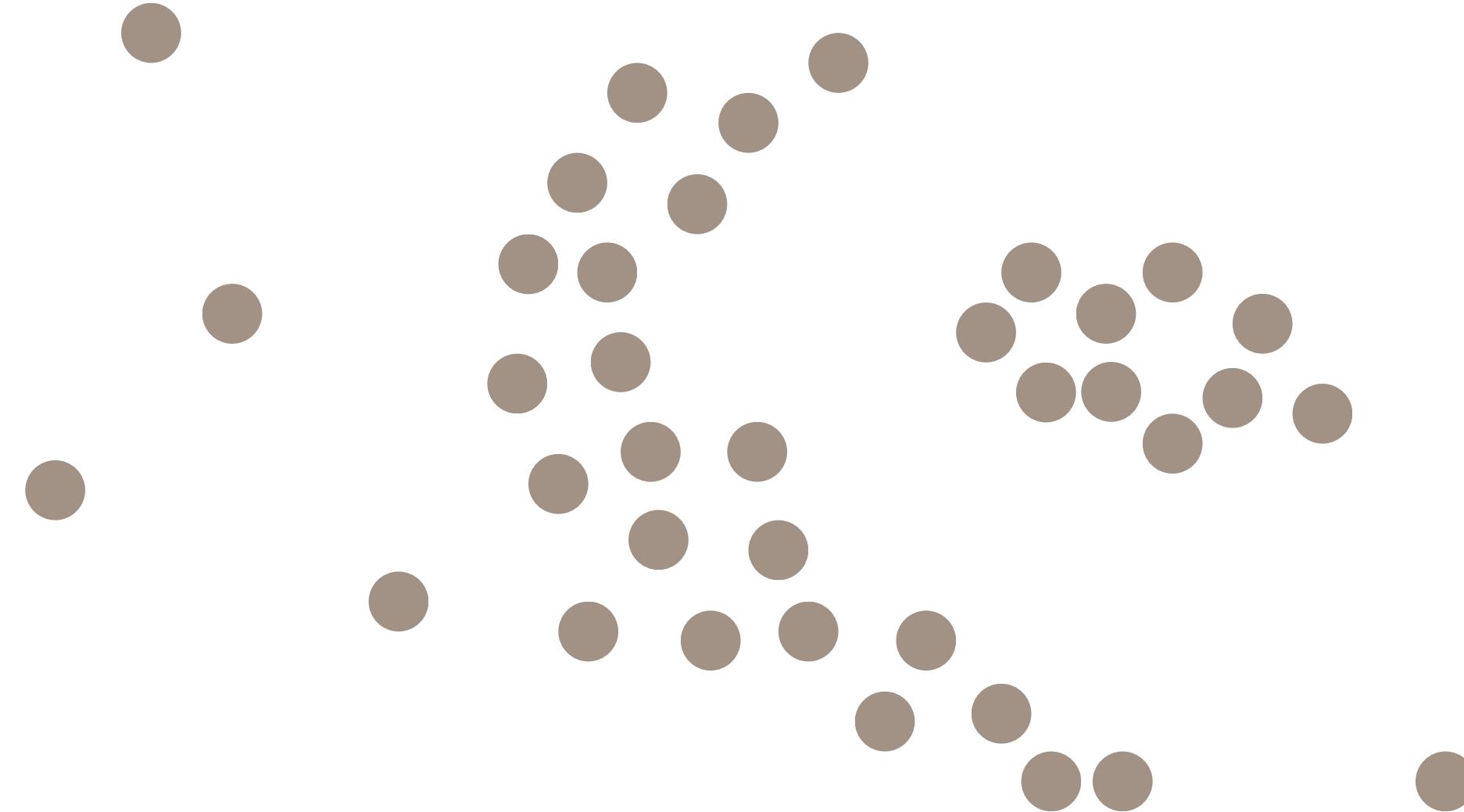
## Parámetros clave

- **$\epsilon$  (eps):** radio de vecindad.
- **minPts:** número mínimo de puntos necesarios en un radio  $\epsilon$  para ser considerado.

Con esto, **los puntos se clasifican** como:

1. **Núcleo (core point):** tiene al menos **minPts** vecinos dentro del radio  $\epsilon$ .
2. **Borde (border point):** está dentro de la vecindad  $\epsilon$  de un punto núcleo, pero no cumple por sí mismo con minPts.
3. **Outlier:** no es núcleo ni borde (queda fuera de clusters).

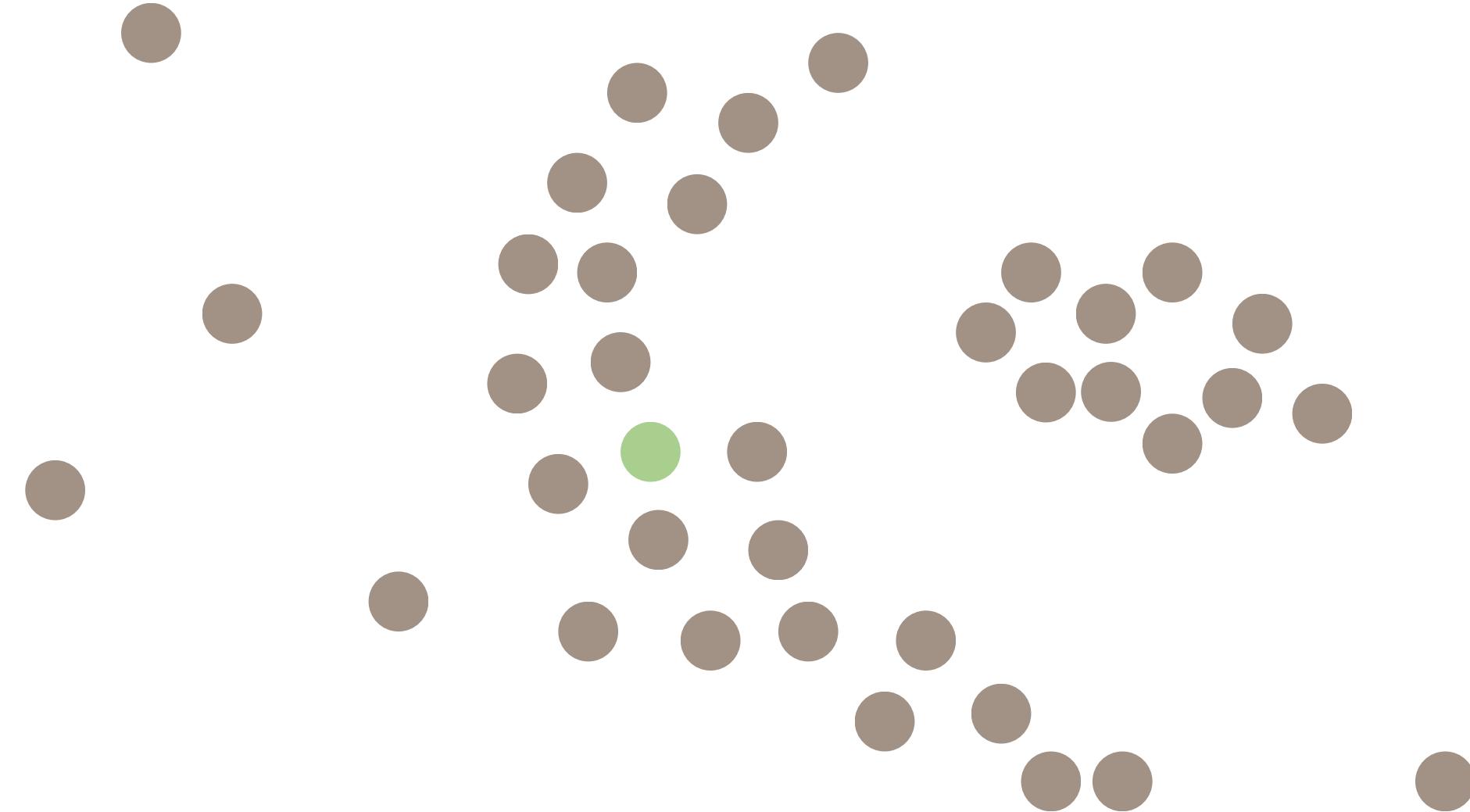
# DBSCAN



Vamos a clusterizar este conjunto de puntos con DBSCAN

# DBSCAN

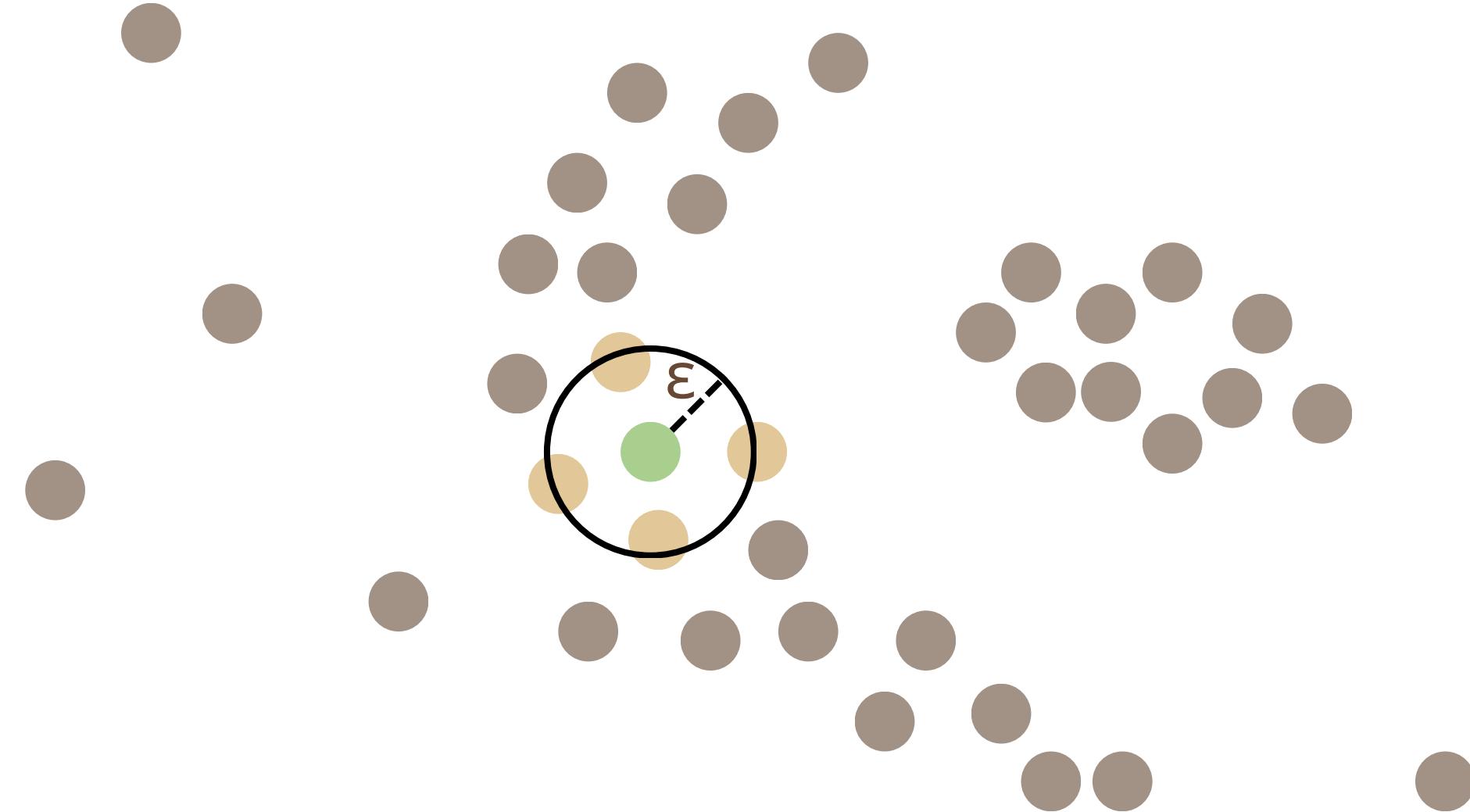
**minPts = 4**



Primero contamos la cantidad de puntos más cercanos a cada punto, si la cantidad de puntos es mayor a igual a **minPts** entonces ese es un **Core Point**

# DBSCAN

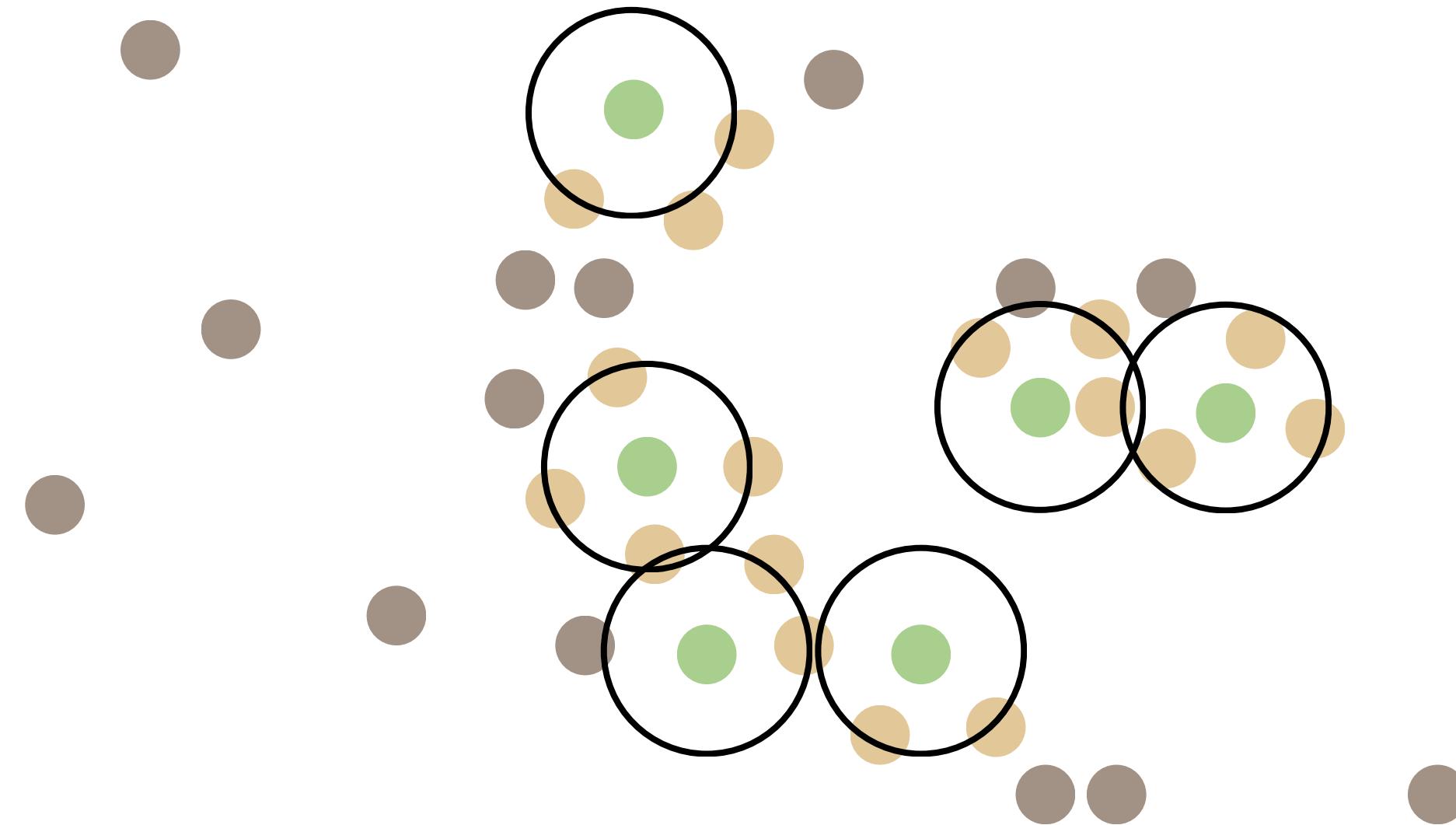
**minPts = 4**



Primero contamos la cantidad de puntos más cercanos a cada punto, si la cantidad de puntos es mayor a igual a **minPts** entonces ese es un **Core Point**

# DBSCAN

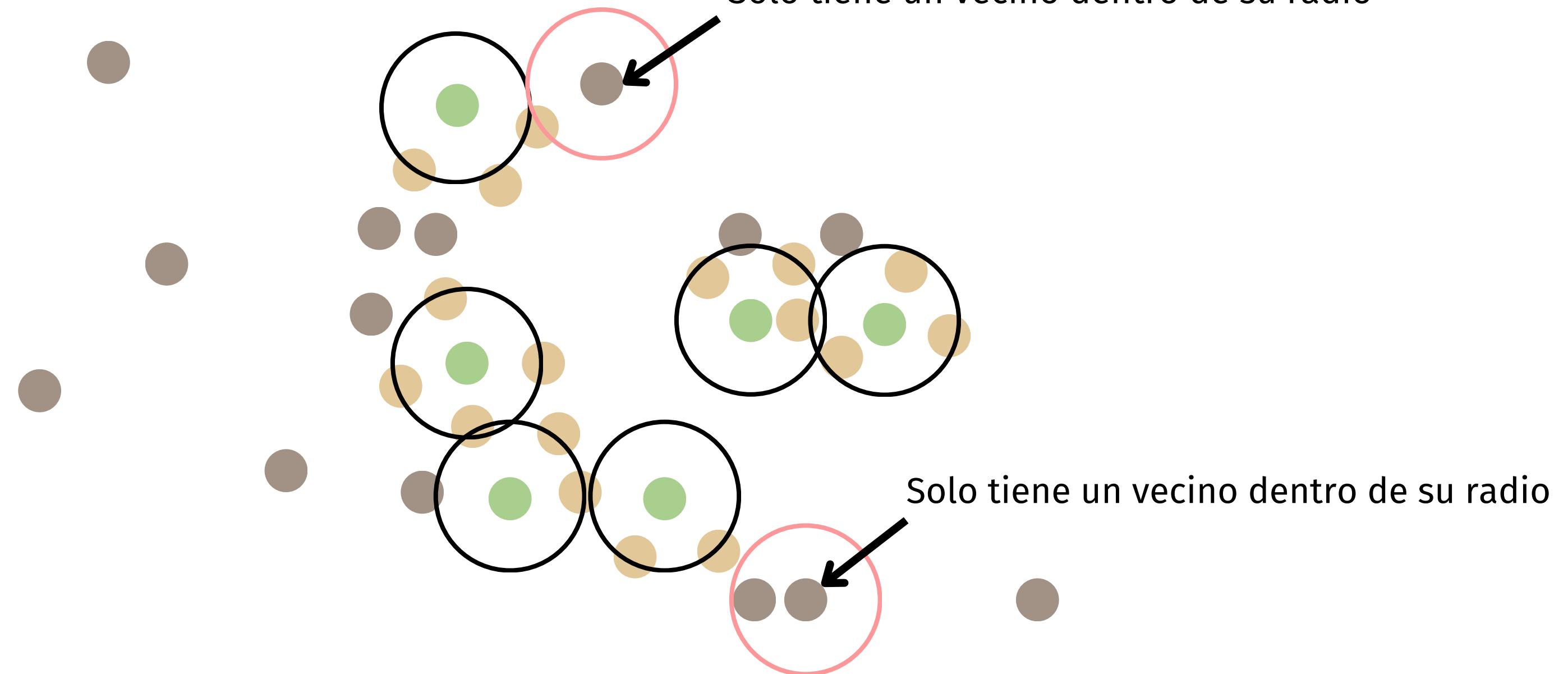
**minPts = 4**



Primero contamos la cantidad de puntos más cercanos a cada punto, si la cantidad de puntos es mayor a igual a **minPts** entonces ese es un **Core Point**

# DBSCAN

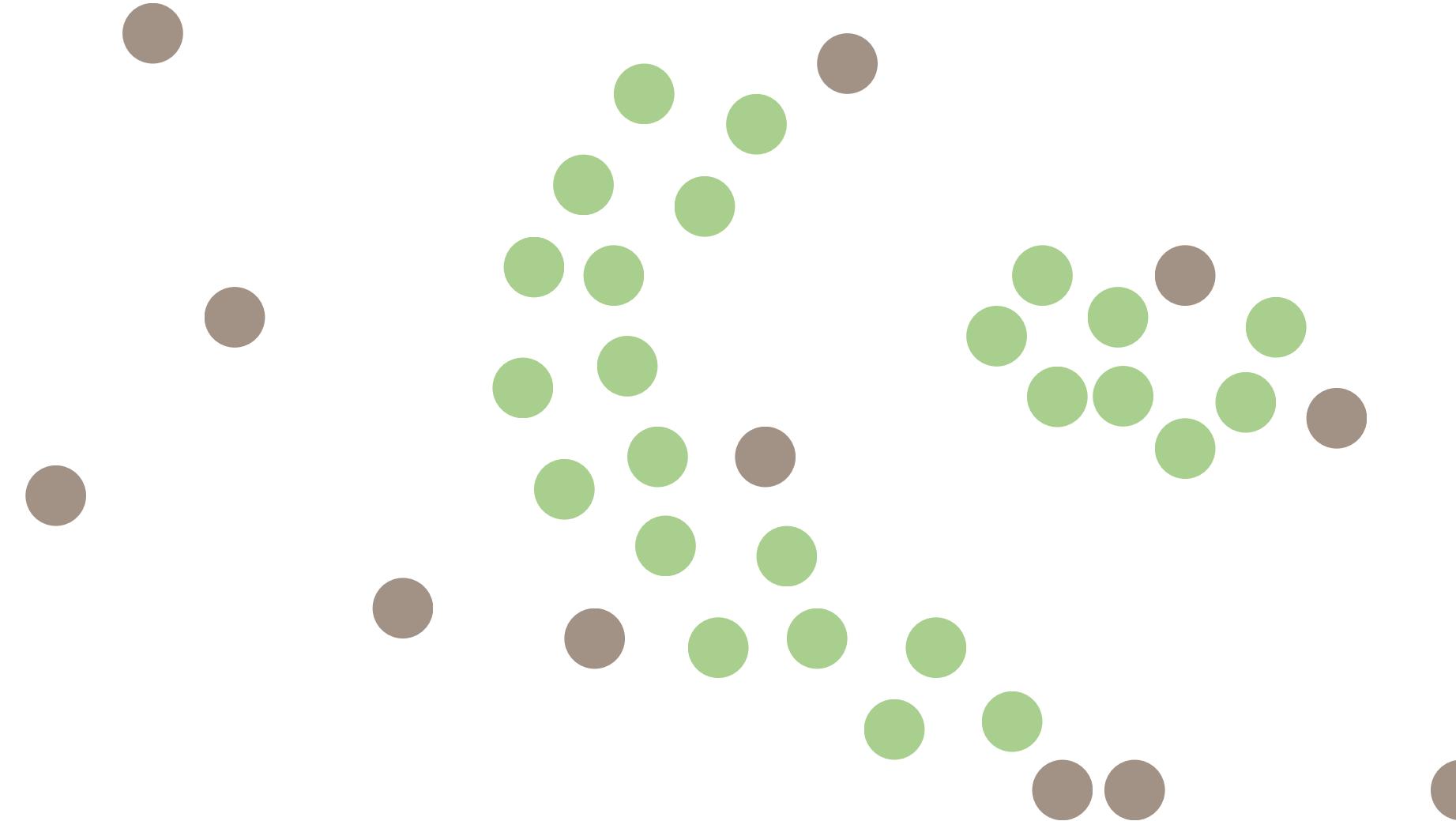
**minPts = 4**



Primero contamos la cantidad de puntos más cercanos a cada punto, si la cantidad de puntos es mayor a igual a **minPts** entonces ese es un **Core Point**

# DBSCAN

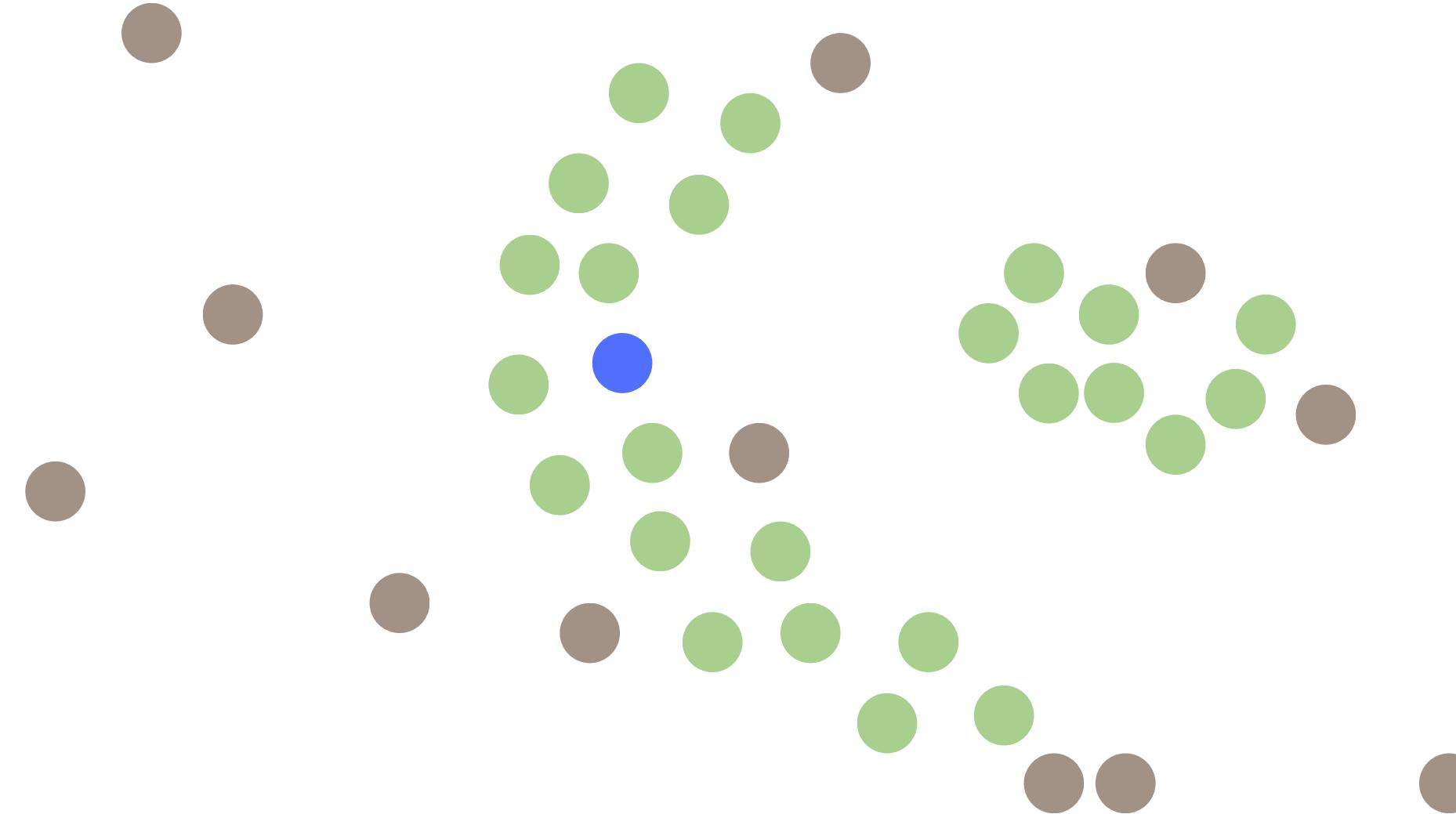
**minPts = 4**



Los puntos verdes son todos los **Core Points**, estos puntos todavía están asignados a ningún cluster, solo cumplen con el criterio de tener al menos **MinPts** vecinos

# DBSCAN

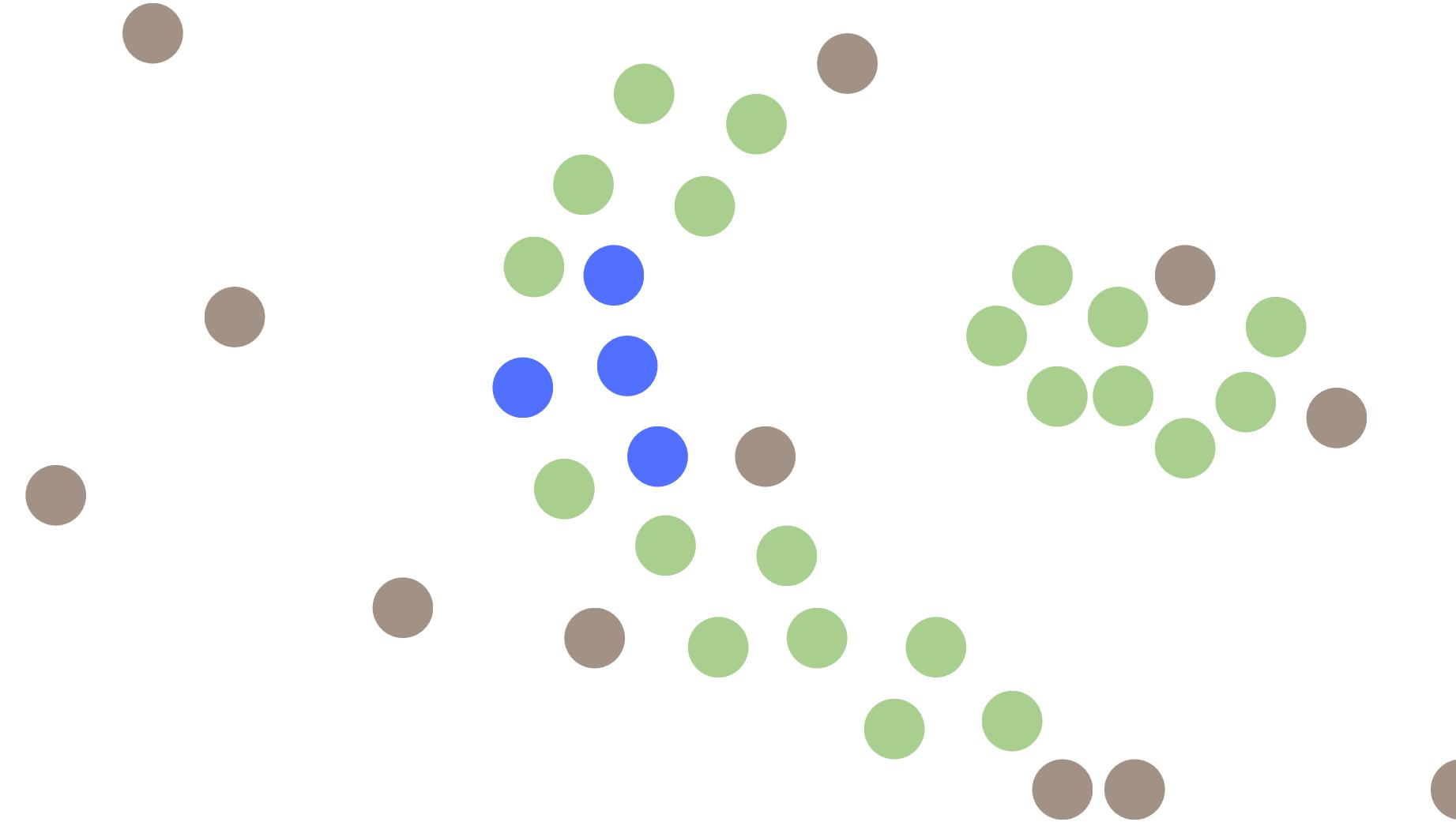
**minPts = 4**



Escogemos un **Core Point** al azar, ese va a ser el primer punto del **primer cluster**.  
Desde ese punto, vamos a agregar al cluster los puntos cercanos a cualquier punto  
que ya esté dentro del cluster

# DBSCAN

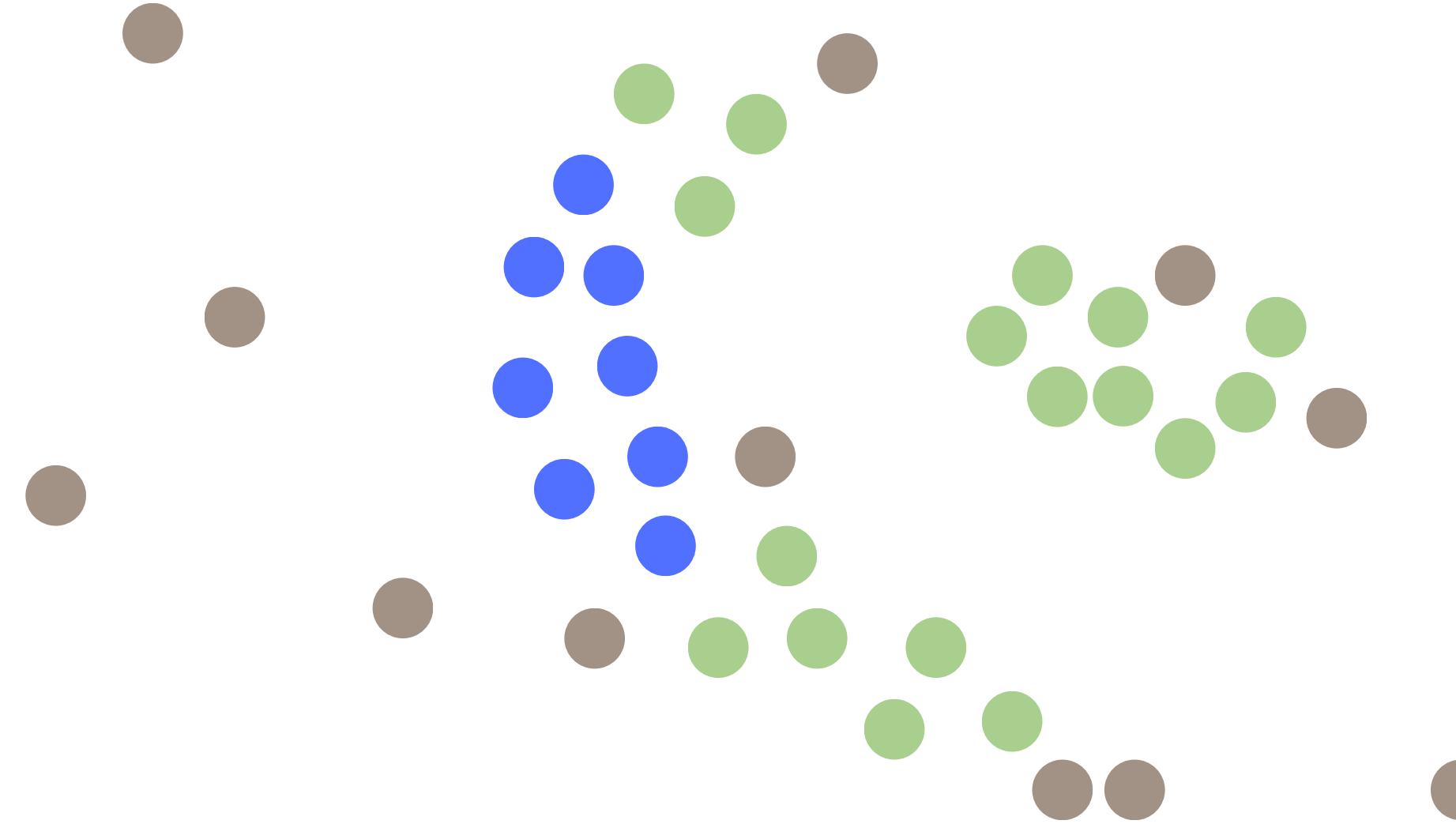
**minPts = 4**



Escogemos un **Core Point** al azar, ese va a ser el primer punto del **primer cluster**.  
Desde ese punto, vamos a agregar al cluster los puntos cercanos a cualquier punto  
que ya esté dentro del cluster

# DBSCAN

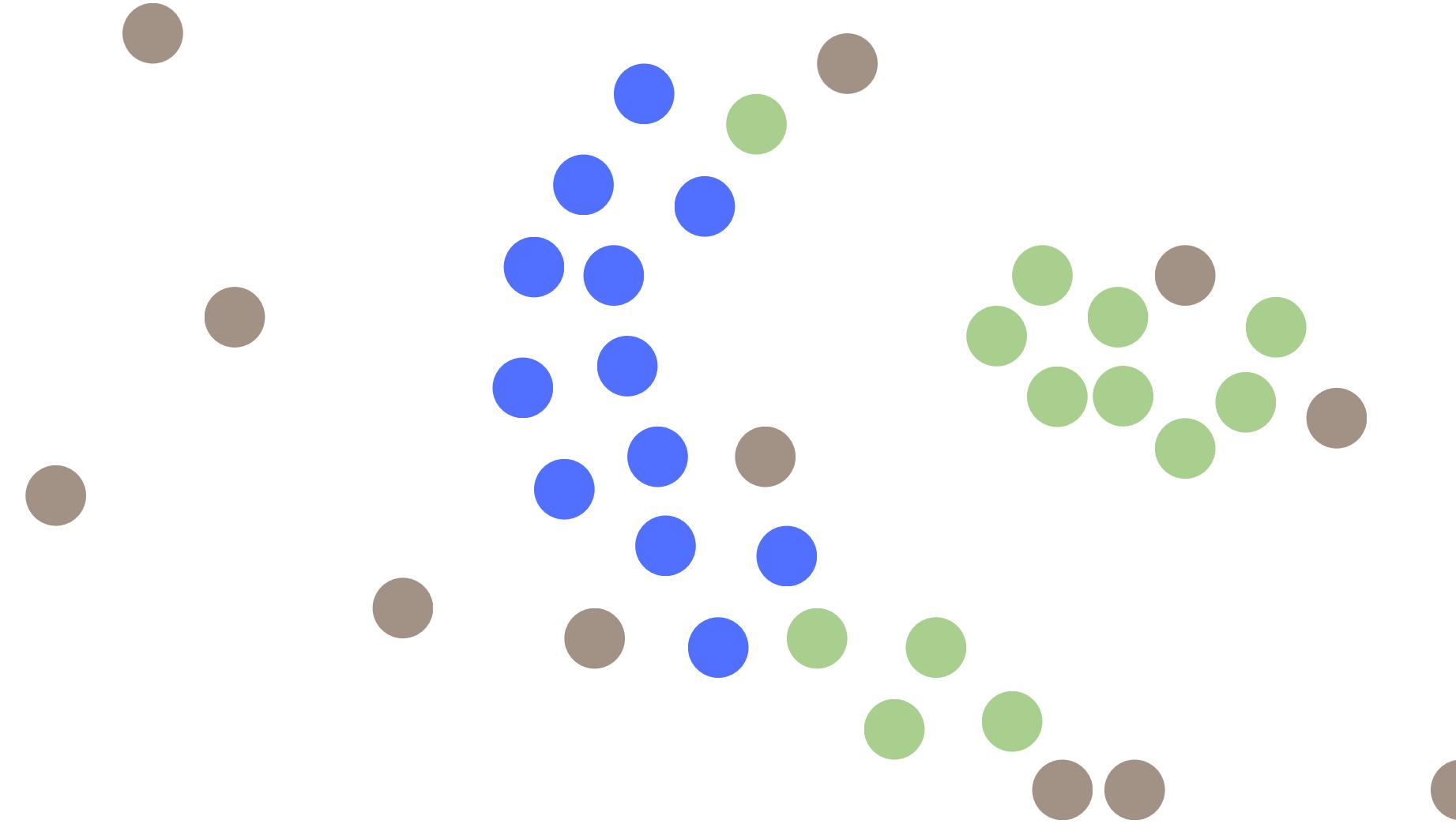
**minPts = 4**



Escogemos un **Core Point** al azar, ese va a ser el primer punto del **primer cluster**.  
Desde ese punto, vamos a agregar al cluster los puntos cercanos a cualquier punto  
que ya esté dentro del cluster

# DBSCAN

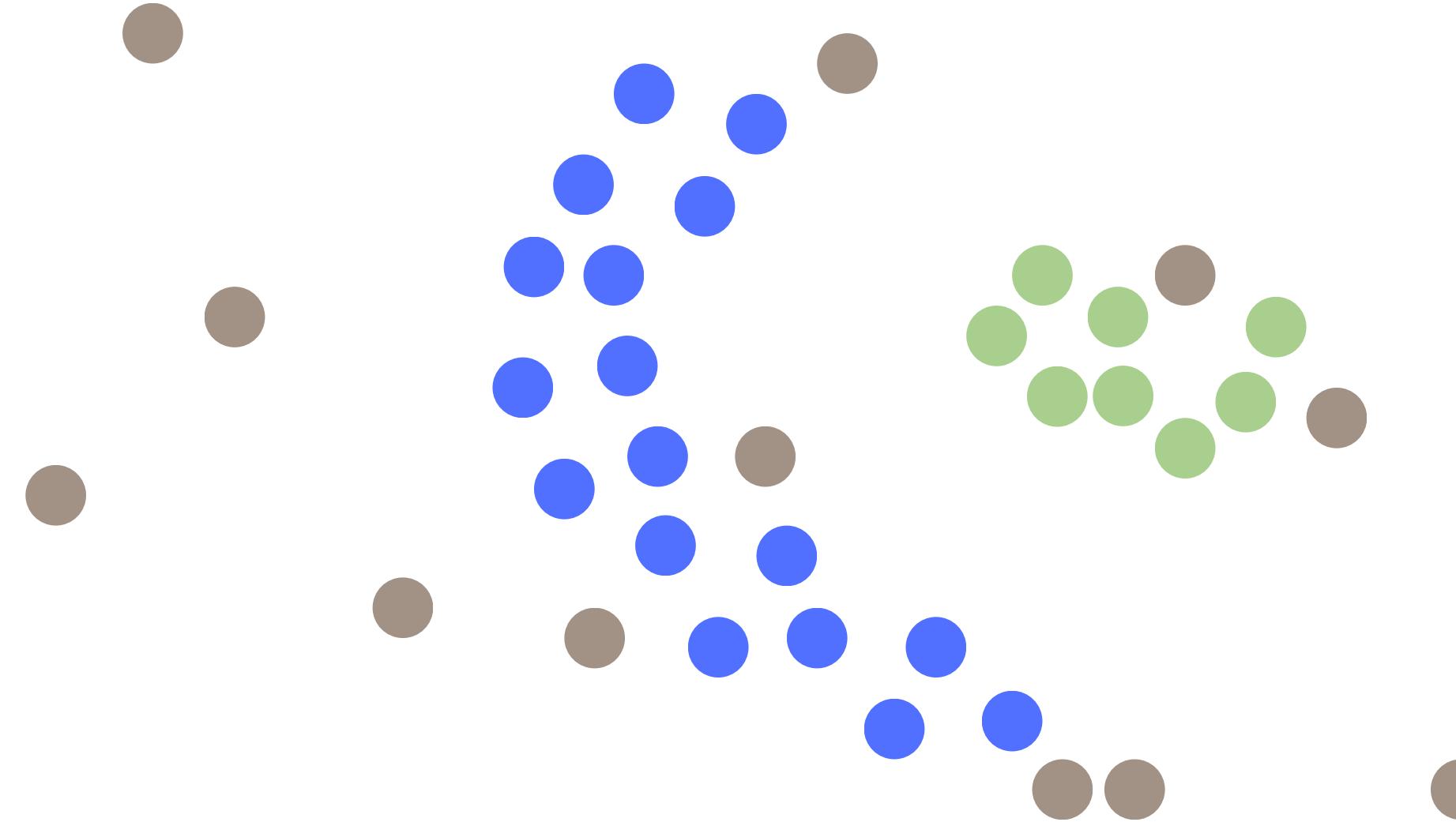
**minPts = 4**



Escogemos un **Core Point** al azar, ese va a ser el primer punto del **primer cluster**.  
Desde ese punto, vamos a agregar al cluster los puntos cercanos a cualquier punto  
que ya esté dentro del cluster

# DBSCAN

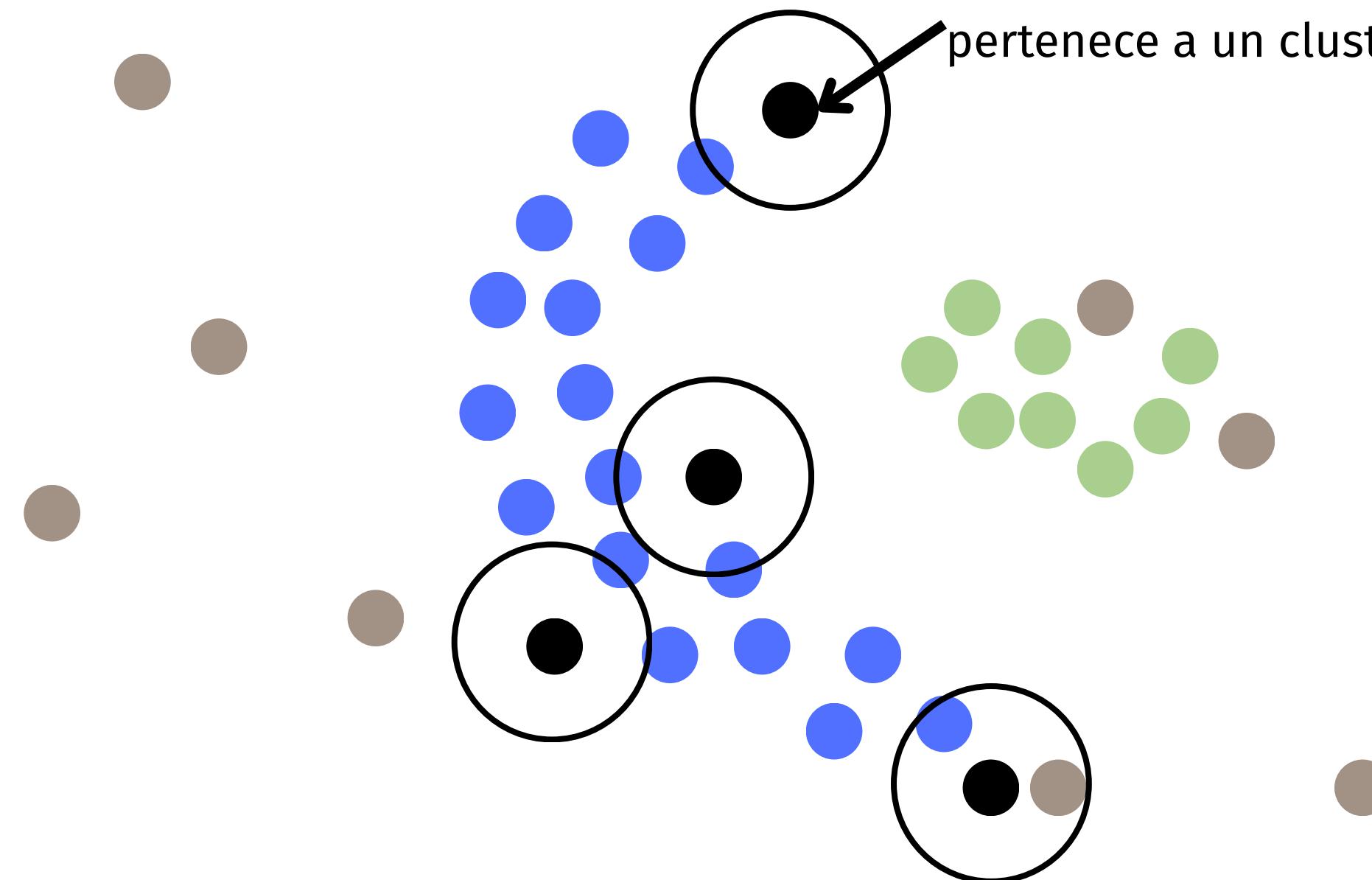
**minPts = 4**



Escogemos un **Core Point** al azar, ese va a ser el primer punto del **primer cluster**.  
Desde ese punto, vamos a agregar al cluster los puntos cercanos a cualquier punto  
que ya esté dentro del cluster

# DBSCAN

**minPts = 4**

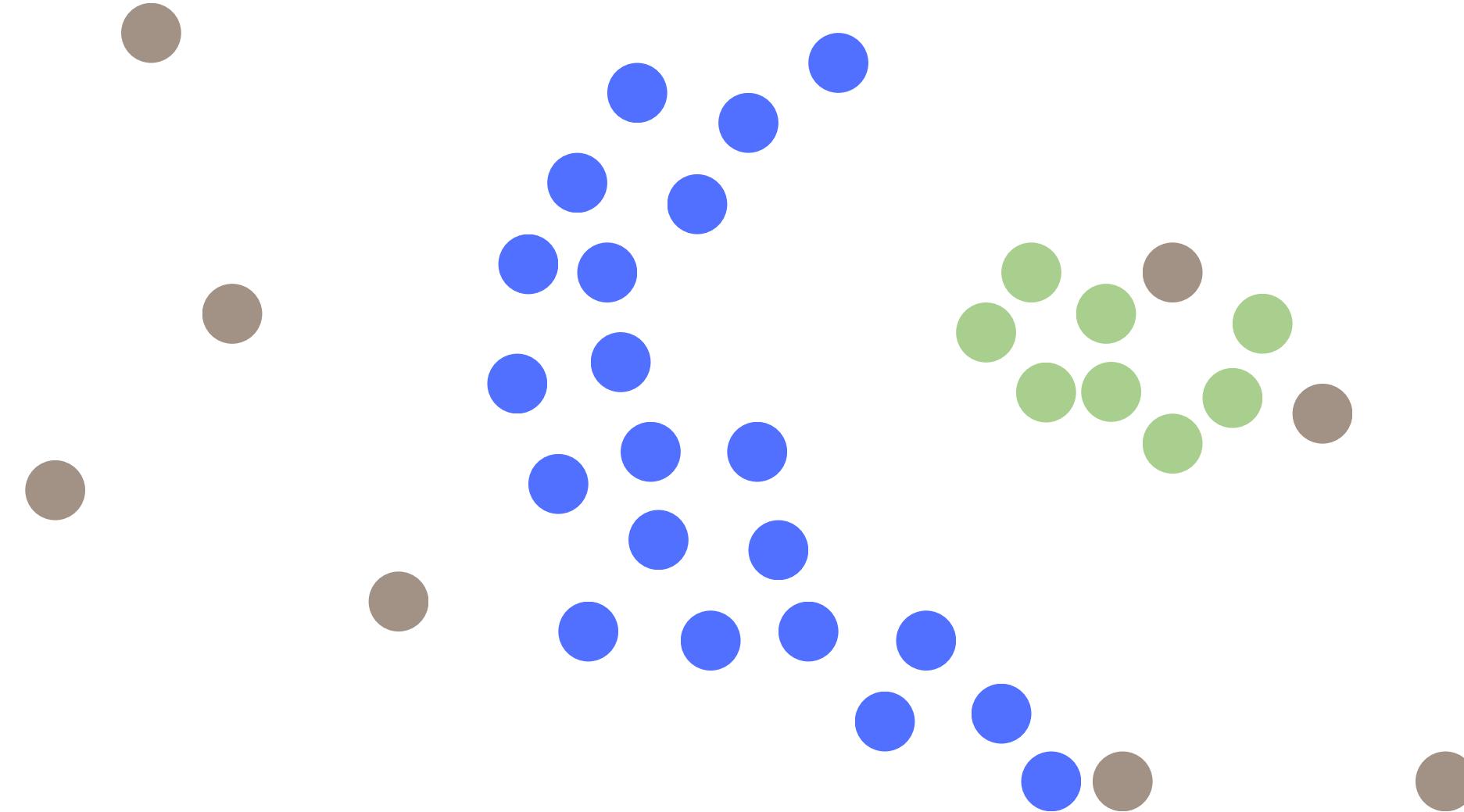


Se agregan todos los Border Point al cluster

Es un **Border Point**, porque hay un punto que pertenece a un cluster (azul) dentro de su radio

# DBSCAN

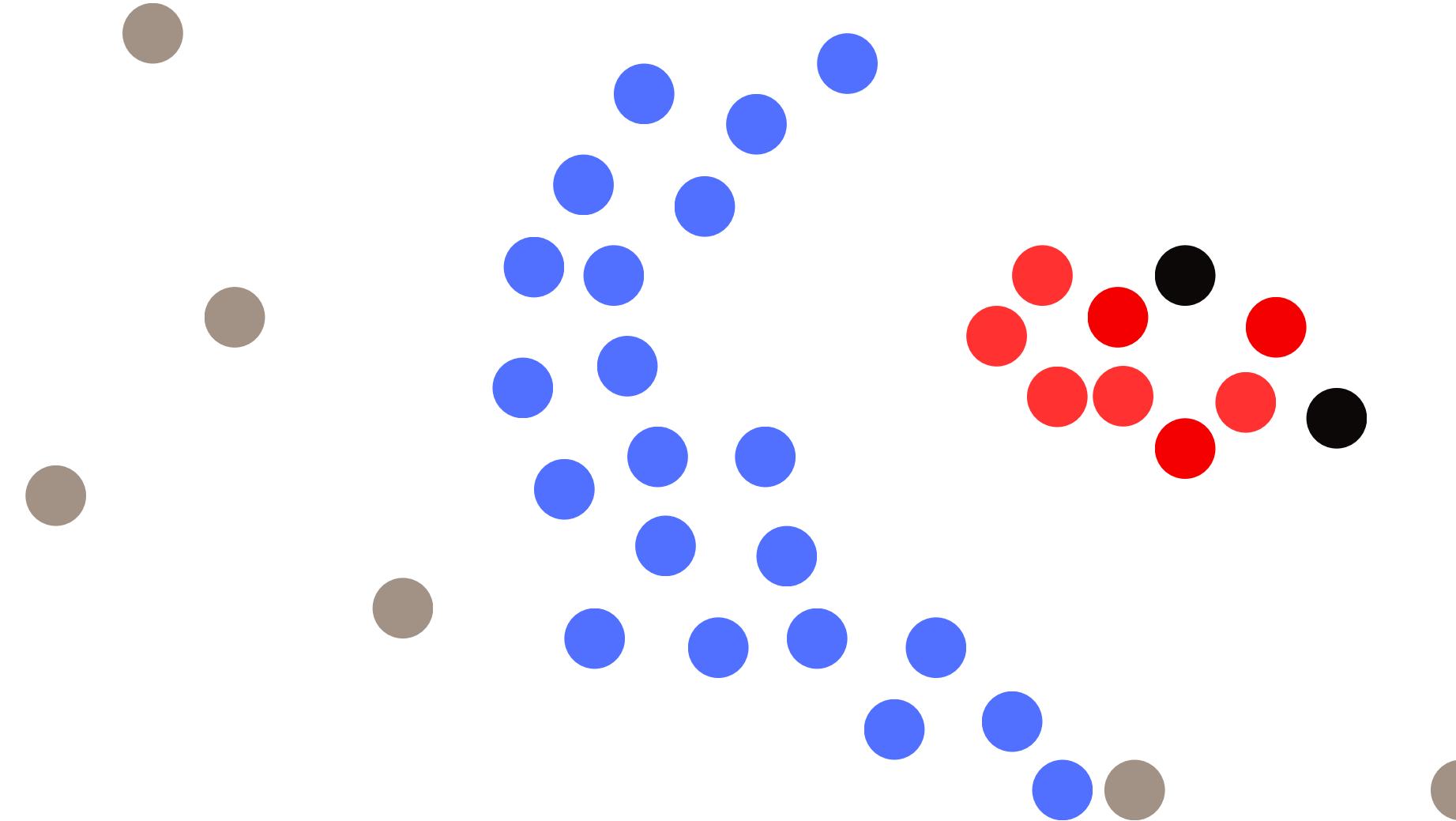
**minPts = 4**



Se agregan todos los Border Point al cluster

# DBSCAN

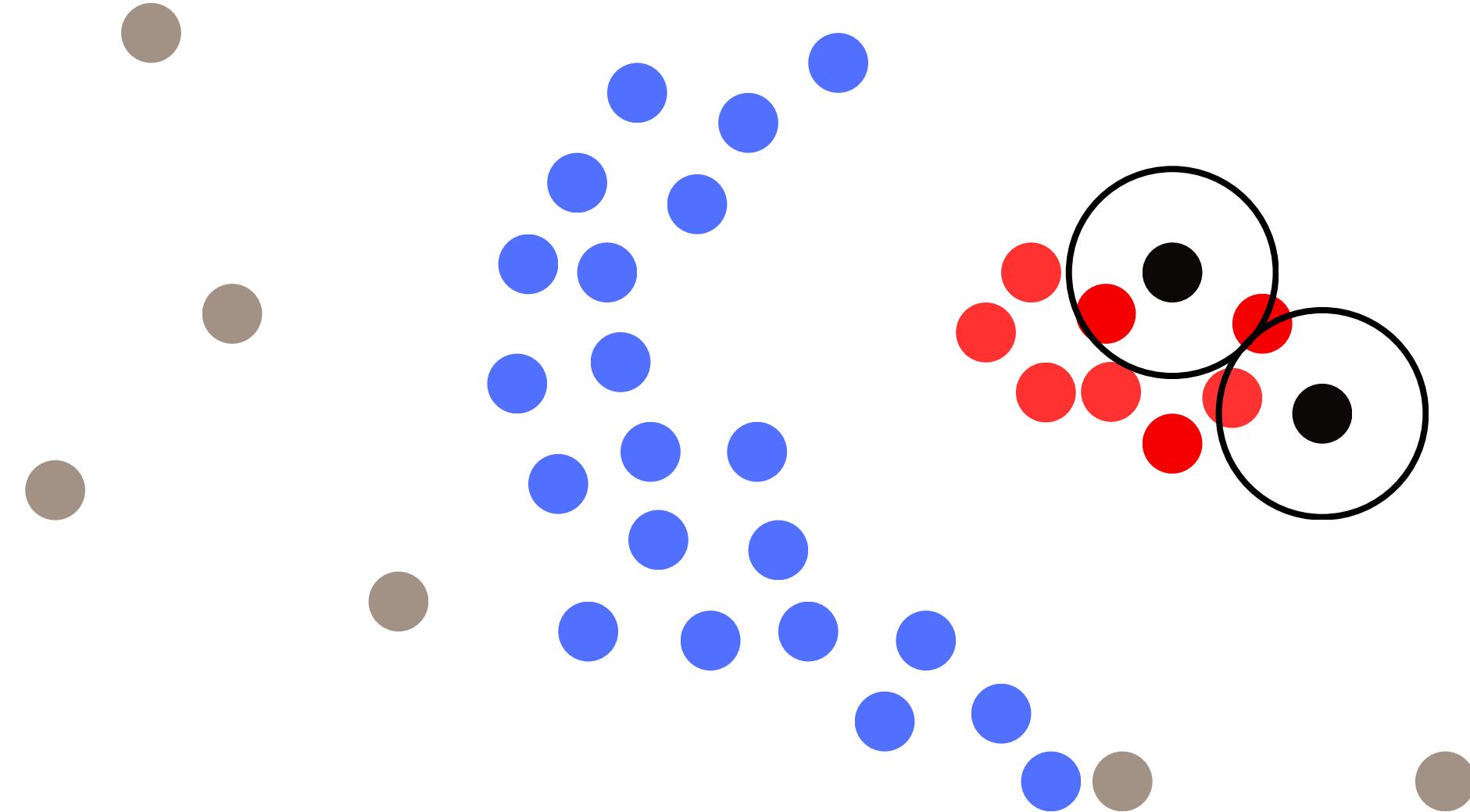
**minPts = 4**



Una vez que ya no se pueden agregar más puntos al cluster, se escoge otro **Core Point** al azar para repetir el proceso

# DBSCAN

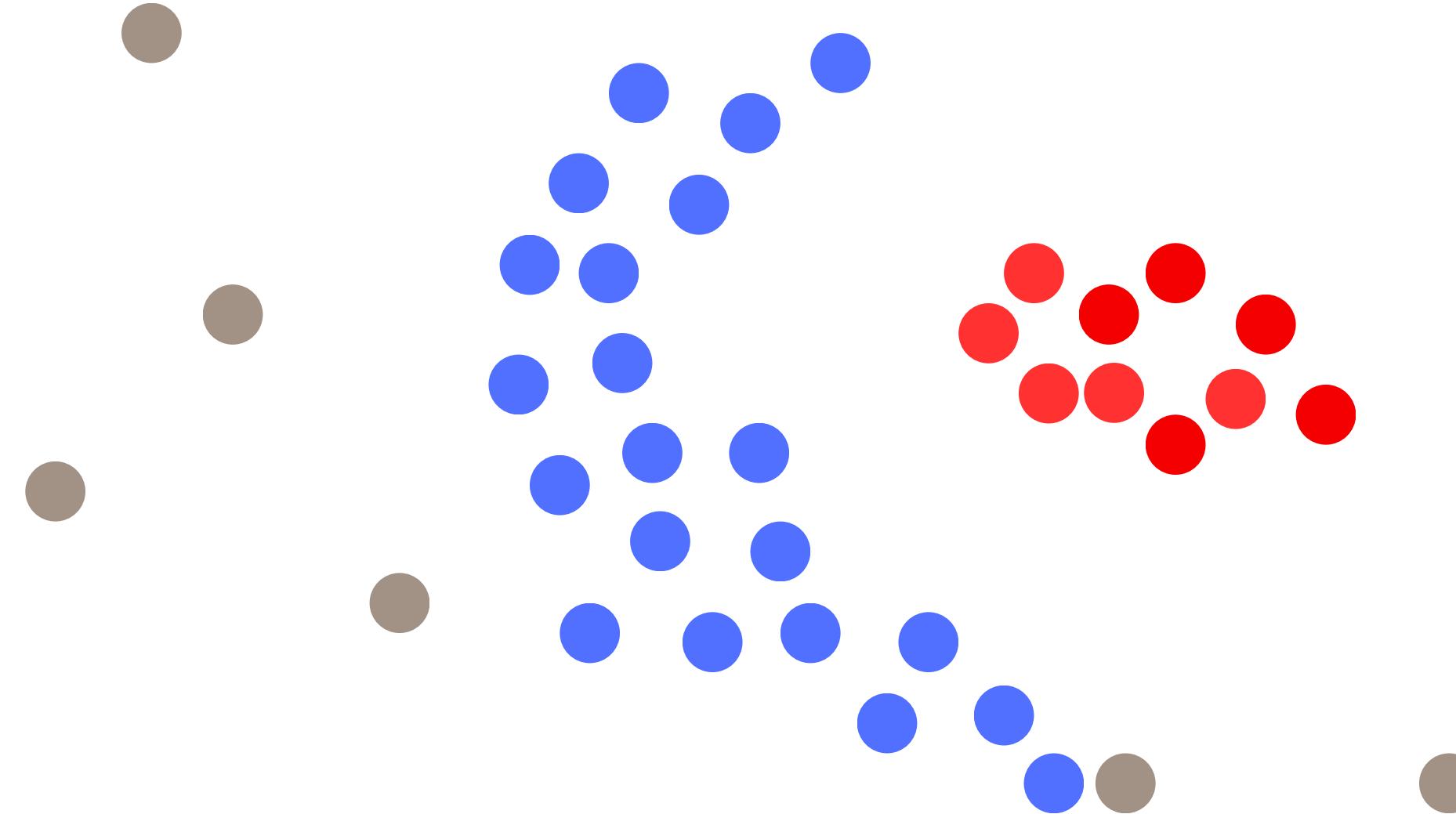
**minPts = 4**



Una vez que ya no se pueden agregar más puntos al cluster, se escoge otro **Core Point** al azar para repetir el proceso

# DBSCAN

**minPts = 4**

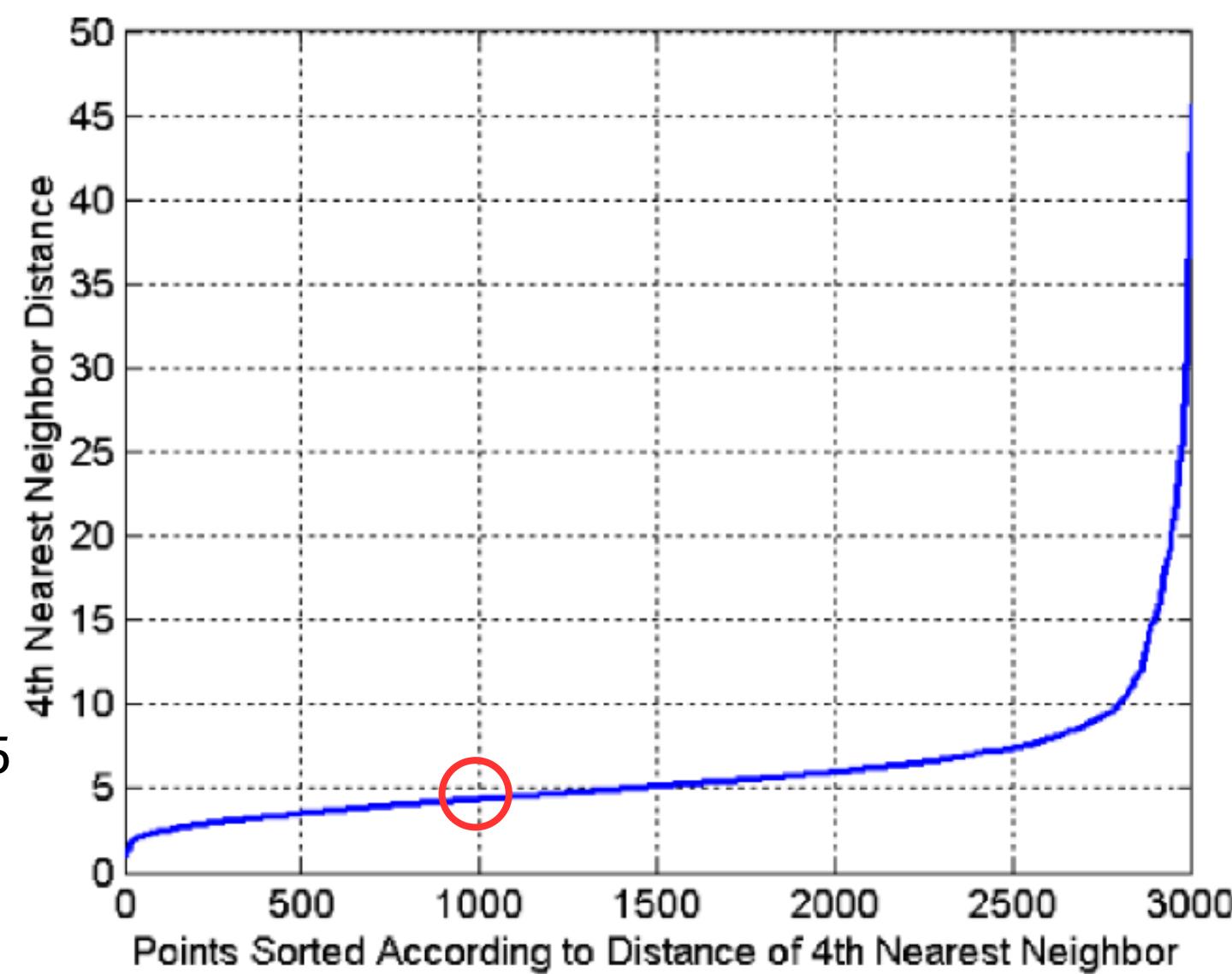


Finalmente nos quedamos con dos cluster, sin definir previamente la cantidad de clusters (k). Los puntos restantes son outliers

# Gráfico k-dist

El k-dist plot nos permite **definir el valor de  $\epsilon$** :

1. Para cada punto, calcular la distancia al minPts-ésimo vecino más cercano.
2. Ordenar esas distancias de menor a mayor.
3. El valor de  $\epsilon$  se elige en la zona donde aparece un codo en la curva

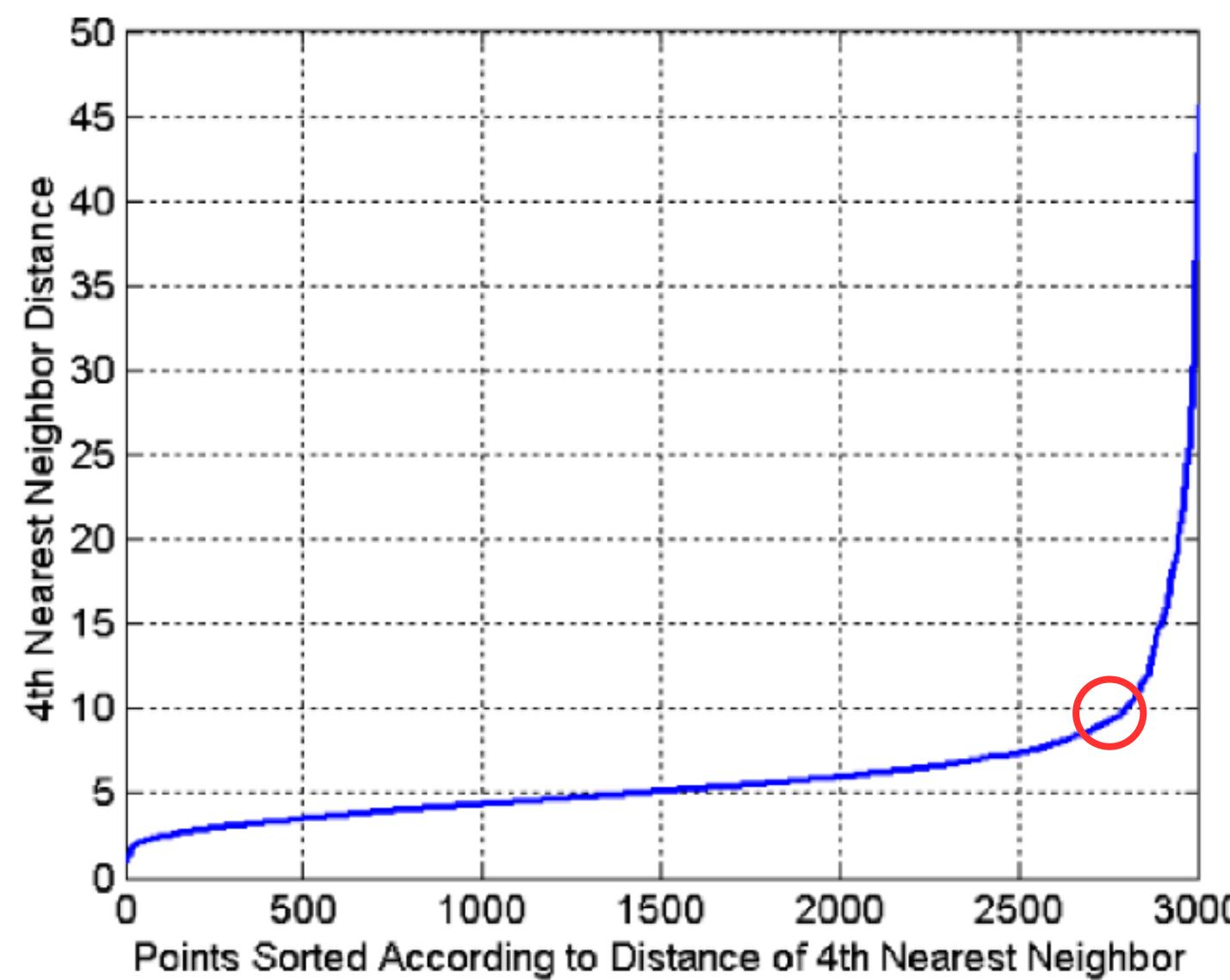


Aprox. 1000 nodos tienen distancia = 5  
a su cuarto vecino

# Gráfico k-dist

El k-dist plot nos permite **definir el valor de  $\epsilon$** :

1. Para cada punto, calcular la distancia al minPts-ésimo vecino más cercano.
2. Ordenar esas distancias de menor a mayor.
3. El valor de  $\epsilon$  se elige en la zona donde aparece un codo en la curva



$\epsilon$  pequeños → solo núcleos muy densos  
 $\epsilon$  grandes → clusters muy amplios

# DBSCAN

```
from sklearn.neighbors import NearestNeighbors
from sklearn.cluster import DBSCAN
from collections import Counter

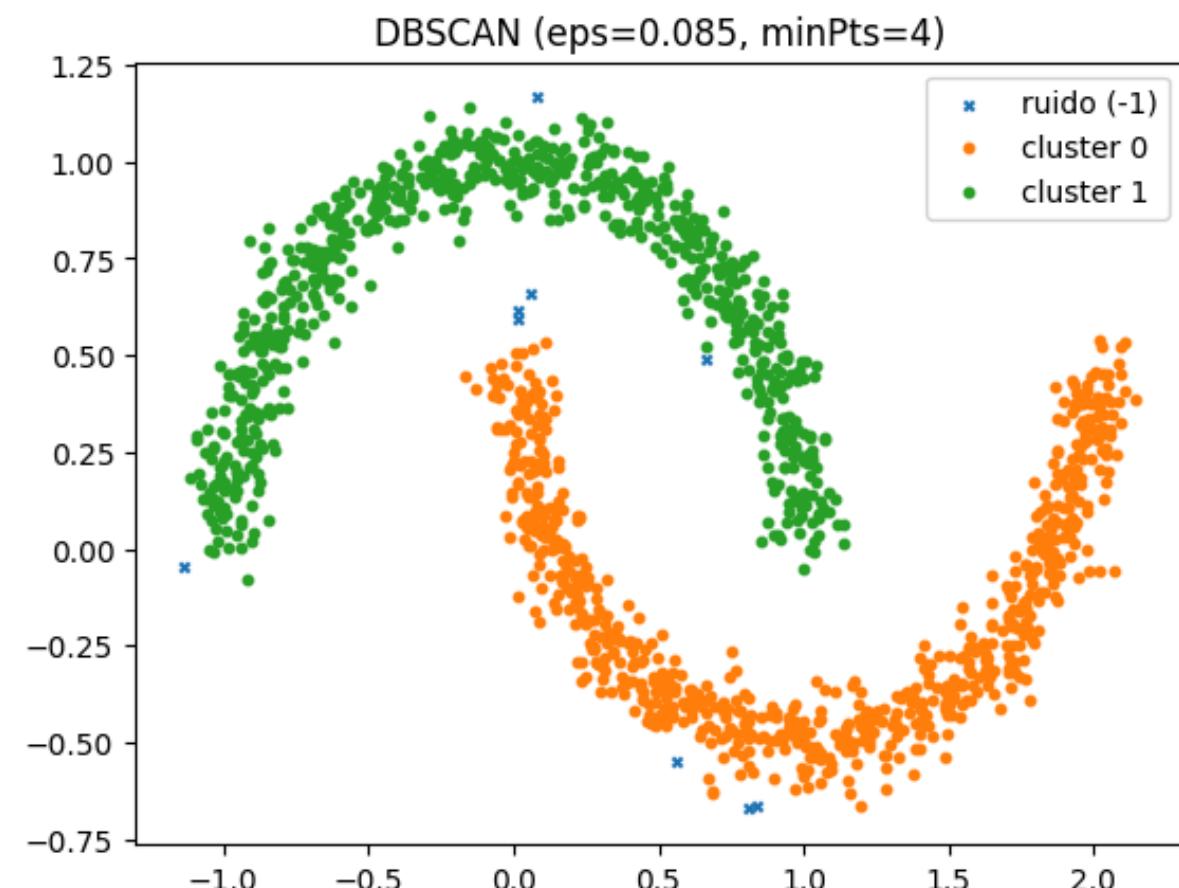
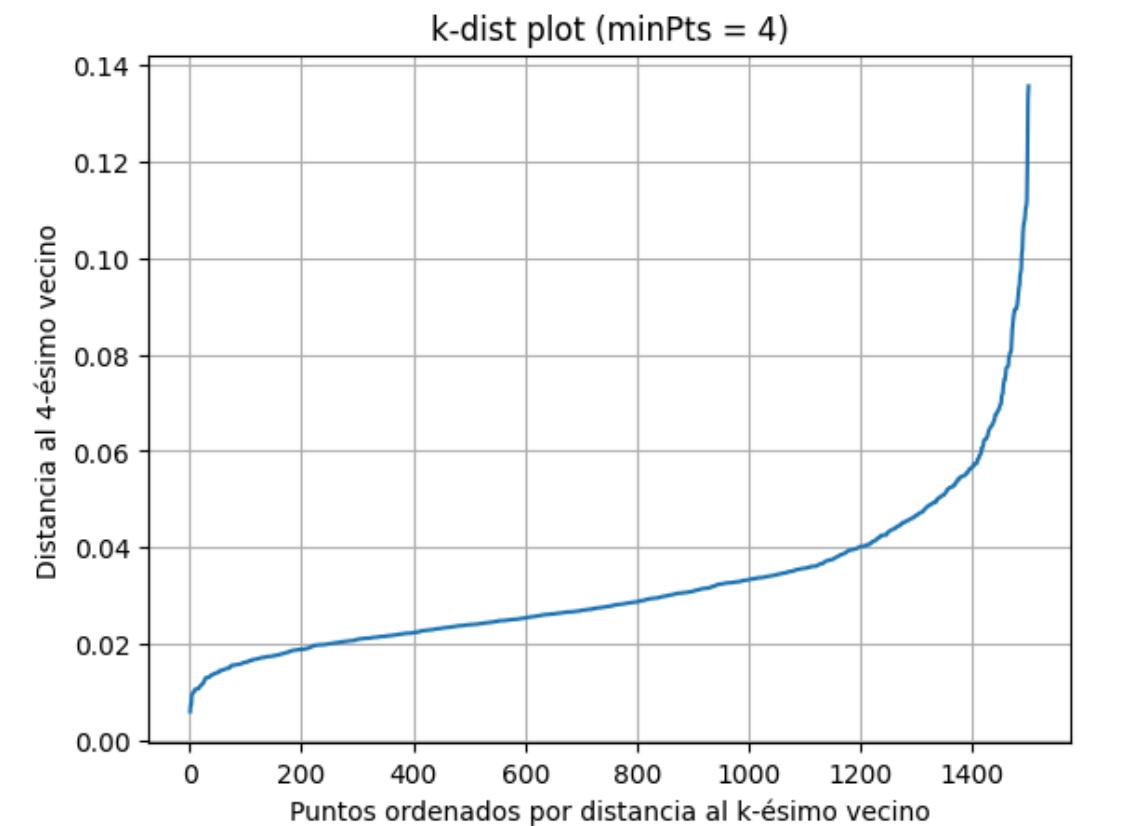
X, _ = make_moons(n_samples=1500, noise=0.06)

# Elegir el "eps"
minPts = 4

# Distancias al k-ésimo vecino más cercano
nn = NearestNeighbors(n_neighbors=minPts).fit(X)
distances, _ = nn.kneighbors(X)
# La distancia relevante es la del minPts vecino
k_dists = np.sort(distances[:, -1])
...
plt.show()

# DBSCAN
db = DBSCAN(eps=best_eps, min_samples=minPts)
labels = db.fit_predict(X)

# Para graficar
counts = Counter(labels)
print("Tamaño por cluster (label -> n):", dict(counts))
print("Ruido (label -1):", counts.get(-1, 0))
...
plt.show()
```



# MINERÍA DE DATOS

**Maximiliano Ojeda**

[muojeda@uc.cl](mailto:muojeda@uc.cl)

---



IIC-2433