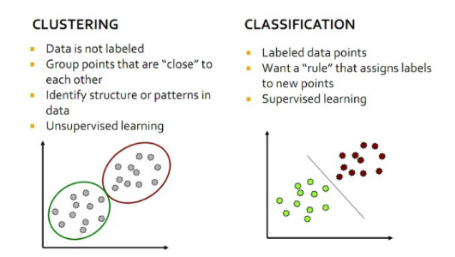
**Clustering**

En el **Aprendizaje No Supervisado** solamente disponemos de la matriz de Features X. **No hay variable target** contrala cual evaluar los resultados. Acá lo que se busca es encontrar regularidades entre los datos que nos permitan agruparlos según su similaridad.

Hay dos grandes grupos de algoritmos dentro de lo que es aprendizaje no supervisado:

1. **Clustering**. Proceso de organizar los objetos en grupos con miembros similares y grupos distintos entre sí. A cada colección de datos similares entre sí la llamamos **cluster**.
2. **Reducción de Dimensionalidad**. Transforman datos de alta dimensionalidad en una representación con menos features que resume las principales características del Dataset.

Mientras que en **clasificación** agrupamos de acuerdo a un **conjunto de datos predefinidos por la clase** (aprendizaje supervisado); en **clustering** agrupamos los **datos con características similares**. Las **clases se descubren** (Aprendizaje No Supervisado).

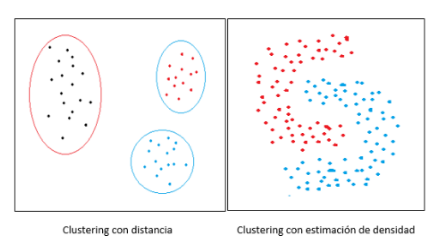


El Clustering **se puede aplicar en:**

* **Marketing:** Grupos de Clientes con comportamiento similar.
* **Biología**: Clasificación de Plantas y Animales.
* **Urbanismo**: Identificar grupos de casas según su tipo de vivienda, valor y ubicación geográfica.
* **Textos**: Agrupar documentos de temas similares.
* **Búsqueda de Outliers**, valores atípicos.
* **Hacer un EDA (Exploratory Data Analysis)** previo a hacer un modelo de Clasificación.

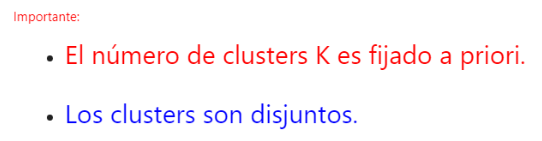
Los resultados del Clustering dependerá de cómo midamos la similaridad de los grupos de datos.

* La **distancia** es la **medida más común**. Se usa en los algoritmos **k-means** y en **Clustering Jerárquico**.
* La **Estimación de Densidad** es agrupar determinando cómo se distribuyen las observaciones en el espacio de dimensiones. La usa el algoritmo DBSCAN.



Si usamos **algoritmos en distancia**, las observaciones se agruparán de forma tal que las **distancias internas** deberán ser **pequeñas** (**dentro del Cluster**), mientras que las **distancias externas** deberán ser **grandes** (**fuera del Cluster**).

**K-Means**: Es uno de los algortimos más populares para identificar Clusters. Debido a su **Simplicidad** (fácil de entender e implementar) y **Eficiencia** (la complejidad en **tiempo de cómputo** es O(tkn), siendo n el tamaño de la muestra, k el número de clusters y t el número de iteraciones; dado que k y t son valores pequeños, se lo considera un *algoritmo lineal*).



Se **agrupan los datos** **en función de su cercanía a** ciertos puntos en el espacio llamados **centroides**.



Funcionamiento del Algoritmo:

1. Se eligen al azar los k puntos en el espacio (centroides).
2. Se asocia cada punto al número de centroide más cercano, definiendo de esta manera los k-cluster.
3. Se redefinen los centroides como los centros geográficos de los puntos dentro de cada cluster. Es decir, la posición nueva de cada centroide es el promedio de las posiciones de los puntos asignados a cada cluster.
4. Repite pasos 2 y 3 en loop hasta que el *algoritmo converge*, es decir,la composición de los clusters ya no cambia, o hasta que se llegue al límite de iteraciones.

El objetivo del algoritmo es minimizar la función objetivo:



Siendo cj el centroide del cluster sj. La Función J es la suma de las distancias al cuadrado de los puntos al centroide de su cluster. Como depende bastante de la selección inicial de los centroides, k-means **no siempre selecciona el mínimo** de la función objetivo.

**En Python:**

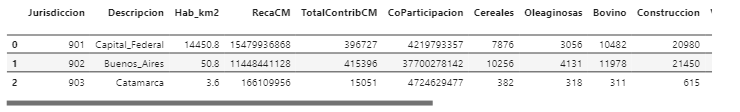
# Vamos a intentar agrupar las provincias Argentinas:

df = read.csv(‘../Data/provincias\_actividad.csv’, sep=’;’)

print(‘Filas:’, df.shape[0], ‘Columnas:’, df.shape[1])



df.head(3)



# Al utilizar distancias para calcular los clusters, conviene estandarizar las variables previamente.

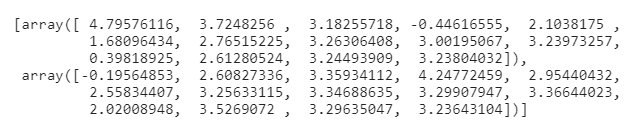
X = df.drop(df.columns[[0,1]], axis = 1) # Eliminamos las variables de texto.

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

np.set\_printoptions(suppress=True)

[X\_scaled[i] for I in range(2)]



# Cuando instanciamos el método, le damos dos hiperparámetros: **n\_clusters** (cantidad de clusters a generar); **n\_init** (número máximo de iteraciones; puede que termine antes).

from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters = 7, n\_init = 10, random\_state = 0)

# Ajustamos el modelo a los datos estandarizados.

kmeans.fit(X\_scaled)



# Ahora obtenemos los parámetros del modelo: labels\_ (vector con la asignación del cluster a cada observación) y clusters\_centers\_ (coordenadas de los centroides).

labels = kmeans.labels\_

centroids = kmeans.cluster\_centers\_

# Dejó solas a Capital Federal (3) y Buenos Aires (1), que son las dos con valores mayores. Después agrupa a Santa Fé y Córdoba(2), que le siguen en valores. Luego agrupa al resto de las provincias.

# Concatenamos la descripción con el Cluster asignado a cada registro.

frames = [df[‘Descripción’],pd.DataFrame(label, columns=[‘Cluster’])]

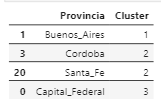
result = pd.concat(frames, axis = 1)

# Cambiamos el nombre de la columna Descripción por Provincia:

clusters =result.rename(columns = {‘Descripción’:’Provincia’})

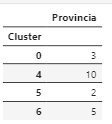
# Miramos qué provincias fueron asignadas a los clusters 1, 2, 3 y 4:

clusters.loc[clusters.Cluster.isin([1,2,3])].sort\_values(‘Cluster’)



# Ahora vamos a ver cuántas provincias hay asignada a cada Cluster:

clusters.loc[clusters.Cluster.isin([0,4,5,6])].groupby(‘Cluster’).count()



# Ahora vamos a ver qué provincias fueron asignadas a los clusters 0, 4, 5 y 6:

clusters.loc[clusters.Cluster.isin([0,4,5,6])].sort\_values(‘Cluster’)



Cómo podemos evaluar si la cantidad K de clusters es la óptima?

K-means busca minimizar la suma de las distancias al cuadrado de los puntos al centroide de su cluster. Vamos a calcular el valor de la función para distintos K, usando el atributo **inertia\_**

**En Python:**

Sq\_distances = []

k\_values = range(2;10)

for k in k\_values:

kmeans = KMeans(n\_clusters = k, n\_init = 10, random\_state = 0)

kmeans.fit(X\_scaled)

sd\_distances.append(kmenas.intertia\_)

plt.figure(figsize = (5,3))

sns.lineplot(x = k\_values, y = sq\_distances, marker = ‘o’, size = 30, legend = False)

plt.ylabel(‘Suma Distancias Cuadráticas’)

plt.xlabel(‘Número de clusters’)

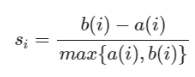


A medida que aumento los clusters, la suma de las distancias cuadráticas se achica. Si k = n, entonces tendríamos un centroide para cada cluster y las distancias cuadrádicas serían todas nulas. Pero lo que nos dice este caso es que cada observación es totalmente distinta de todas las demás, por ende, no estaría agrupando nada.

Con la **regla del codo (Elbow Method)**, podemos aplicar un criterio para elegir el k a partir de la observación del gráfico recién generado. Se elige el **k a partir del cual la curva se aplana** (donde está el codo). En este caso, k = 4 es una buena opción.

**Explicación Coloquial:** Entonces, con **inertia\_** podemos obtener la **suma de errores cuadráticos de cada punto en relación a su centroide,** asignado en función del label al que esté etiquetado dicho punto en el momento en el que lo analizamos. Si vemos la **evolución de esta suma de errores cuadráticos** en función de **k**, podemos **ver gráficamente** a partir de qué k lo que hacemos es complejizar el modelo sin obtener una ganancia significativa de reducción de suma de errores cuadráticos. En el caso extremo en que complejizáramos el modelo a k = n, entonces la suma de errores cuadráticos sería nula, pero el modelo no nos serviría para nada: nos diría que estamos ante observaciones no relacionadas entre sí ninguna de la otra; por eso se hace este análisis visual para poder hacer un trade-off entre complejidad del modelo y ganancia en reducción de error cuadrático.

El **coeficiente Silhouette** mide **cuán cercano** es un **punto al resto** de los de su **mismo cluster**; sobre cuán cercano es a los puntos del clúster más próximo. Entonces para cada punto i podemos obtener un coeficiente si:



Donde a(i) es la diferencia promedio a los otros puntos del mismo cluster y b(i) es la distancia promedio a todos los puntos del cluster vecino más cercano.

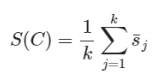
Entonces



* Si s es cercano a 1, entonces eso nos indica que el punto i está correctamente etiquetado.
* Si s es cercano a 0, entonces eso indica que así como está etiquetado al cluster actual, podría etiquetarse al cluster vecino más cercano.
* Si s es cercano a -1, entonces el punto i está mal etiquetado y debería reetiquetarse en el cluster vecino más cercano.

**Explicación Coloquial:** Es decir, tenemos una medida *b* de qué tan cerca está un punto respecto de su cluster vecino más cercano y otra medida *a* respecto de sus otros puntos del mismo cluster. Si la distancia con respecto al vecino más cercano es mucho más grande que la distancia respecto a sus otros puntos del mismo cluster (al punto extremo en el que la distancia intra cluster sea tan baja que pueda asumirse como nula), entonces el coeficiente de Silhouette tenderá a 1 (b – 0 / b = 1). Si se diera lo inverso, (al punto tal en el que la distancia del punto respecto a los puntos de su cluster vecino más cercano sea tan baja que pueda asumirse como nula), tendería a -1 y habría que reclasificar ((0 – a )/ a = -1). Si fueran valores similares (al punto extremo de asumir que son exactamente iguales b = a), tendería a cero y daría lo mismo que esté en uno u otro cluster (b – a = 0, entonces, b-a / max(a,b) = 0 / max (a,b) = 0).

Para obtener un score del clustering, lo que hacemos es promediar los coeficientes:



Siendo j el coeficiente promedio de todos los puntos dentro del cluster j.

**En Python:** Vamos a graficar el score en función de k. Veremos que el valor óptimo se encuentra entre k = 2 y k = 4:

from sklearn.metrics import silhouette\_score, silhouette\_samples

sil = []

k\_values = range(2,10)

for k in k\_values:

kmeans = KMeans(n\_clusters = k, n\_init = 10, random\_state = 0)

kmeans.fit(X\_scaled)

score = silhouette\_score(X\_scaled, kmeans.labels\_)

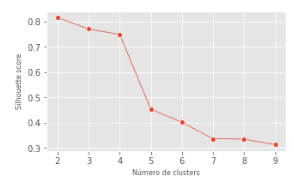
sil.append(score)

plt.figure(figsize = (5,3))

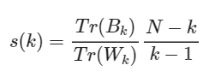
sns.lineplot(x = k\_values, y = sil, marker = ‘o’, size = 30, legend = False)

plt.ylabel(‘Silhouette score’, fontsize = 8)

plt.xlabel(‘Número de clusters’, fontsize = 8)



El **Calinski-Harabasz score** mide la relación entre la dispersión inter-clusters y la dispersión intra-clusters. Una estructura de clusters bien definidos tiene una alta dispersión entre diferentes clusters y una baja dispersión dentro de cada cluster:



Siendo Tr(Bk) y Tr(Wk) las trazas de las matrices de dispersión entre los clusters (Tr(Bk)) e intra clusters (Tr(Wk)).

El factor  tiende a aumentar con el número de clusters, porque al aumentar k, lo que hacemos es generar clusters más chicos y menos dispersos (baja el denominador, entonces aumenta el ratio).

Por otro lado, lo que trata de hacer el segundo factor es contrarrestar un poco el efecto del primero; a mayor cantidad de clusters k, k va tendiendo a N, y este ratio disminuye porque el numerador va tendiendo cada vez más a 0, mientras que el denominador se va haciendo cada vez más grande> .

**En Python:**

# Vamos a calcular este score para distintos valores de K. Según la estructura de los datos nos podríamos encontrar con

* Un pico bien definido, que nos permita elegir el k.
* Caso contrario, aplicar la regla del codo.

from sklearn.metrics import calinski\_harabasz\_score

k\_values = range(2,9)

ch\_scores = []

for k in k\_values:

kmeans = KMeans(n\_clusters = k, n\_init = 10, random\_state = 0)

kmeans.fit(X\_scaled)

score = calinski\_harabasz\_score(X\_scaled, kmans.labels\_)

ch\_scores.append(score)

plt.figure(figsize=(5,3))

sns.lineplot(x=k\_values, y=ch\_scores, marker = ‘o’, size = 30, legend = False)

plt.ylabel(‘Calinski-Harabasz’, fontsize=10)

plt.xlabel(‘Número de clusters’, fontsize = 10)

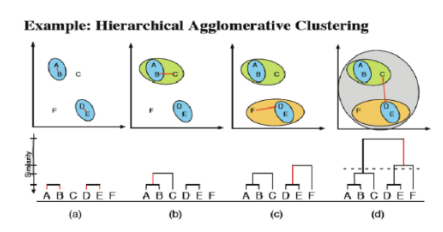


**Resumiendo el modelo K-Means:**

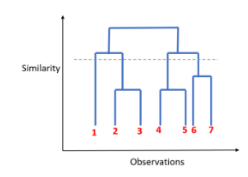
* Hay que **definir** el valor de **K (cantidad de clusters) de antemano**.
* Los **clusters** son **disjuntos**.
* Los **resultados pueden variar** en **función** de la **asignación de** **clusters** aleatoria **inicial**. Pero se puede mejorar haciendo varias corridas y seleccionando el que resulte con menor costo.
* Se basa en la **hipótesis** de que los **clusters** tienen una **geometría esférica**. Por eso **se busca minimizar la distancia euclídea (al cuadrado)**. Puede que no identifique los clusters si tuvieran otra geometría.
* Es **sensible a outliers**: Al asignar todos los puntos a un cluster, los outliers pueden sesgar la estimación de los centroides. En general conviene aislar previamente a los mismos; también puede pasar que se forme un cluster de outliers y que nos terminen ayudando con el análisis de datos.

**Clustering Jerárquico:**

A diferencia de K-Means, el **Clustering Jerarquico parte de** asignar **todos los puntos a su propio cluster**. Cada punto pasa a ser un cluster. A este tipo de algoritmos se los llama **aglomerativos**, o también **bottom-up**. Luego, en cada paso, **construye una jerarquía combinando los dos puntos más cercanos** en un único cluster, hasta terminar con todos los puntos en un único cluster:



Se pueden visualizar los clusters en forma de árbol (**dendograma**). Sirve mucho para determinar el número de clusters más apropiado:

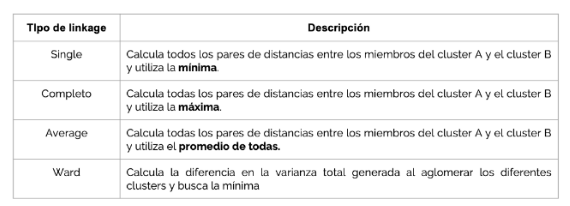


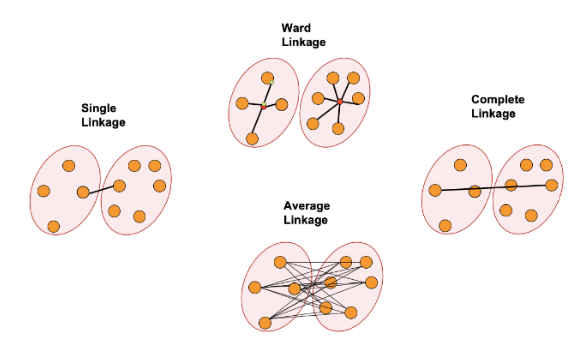
Con la línea punteada horizontal que corta 4 líneas, nos indica que se han seleccionado cuatro clusters. Podemos variar en más o en menos moviendo dicha línea hacia abajo o hacia arriba respectivamente.

A diferencia de K-Means, el Clustering Jerárquico **no necesita que le definamos de antemano la cantidad de grupos K que necesitamos**.

Necesitamos 2 elementos para poder construir el Clustering Jerárquico:

1. Una **medida de distancia**, para determinar la cercanía.
2. Un **criterio para fusionar** los clusters. IE: medir la distancia entre los mismos y seleccionar los más cercanos. A este criterio se lo llama **linkage**.





**Por default**, el modelo trabaja con **Ward**: Busca clusters de dimensiones similares. El modelo funciona bien en la mayoría de los casos.

En caso de tener Clusters con diferentes cantidades de miembros, se puede usar **complete** o **average**.

Con **single linkage** el modelo tiende a generar clusters extendidos en los que las hojas se van agregando de a una.

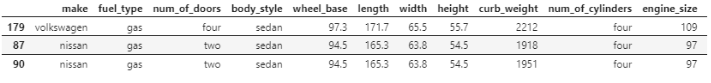
**En Python:** Vamos a generar clusters sobre un DataSet de automóviles.

df = pd.read\_csv(‘../Data/automobile.csv’, sep = ‘;’)

print(‘Filas: ’, df.shape[0],’Columnas: ’, df.shape[1])



df.sample(3)



# Pasamos a seleccionar las features y estandarizar:

columns = [‘wheel\_base’, ’length’, ‘width’, ‘height’, ’horsepower’, ‘highway\_mpg’, ‘price’]

X = df[columns]

Scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Para implementar el modelo, vamos a usar el método **linkage** de la librería scipy. A diferencia de los que vimos hasta ahora, este modelo no tiene los métodos “fit” y “predict”.

from scipy.cluster.hierarchy import dendrogram, linkage, cophenet, fcluster

from scipy.spatial.distance import pdist

Z = linkage(X\_scaled, ‘ward’) # ward indica el tipo de linkage.

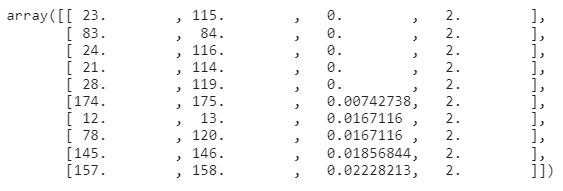
# Linkage nos va a devolver una matriz Z de n-1 filas y 4 columnas:

En cada fila i, correspondiente a la i-ésima iteración, las **dos primeras columnas** indican **los clusters que se combinan** y forman el nuevo cluster n+i. Un cluster con un número menor a n corresponde a una de las n observaciones originales (datos sueltos). La 3er columna indica la **distancia entre los clusters.** La 4ta columna indica el **número de observaciones** a partir del cual el **cluster está formado**.

len(Z)



Z[:10,:] # Las 10 primeras iteraciones.



Z[195:,:] # Las últimas 3 iteraciones:



# Con el método **dendogram** podemos graficar el dendograma a partir del array Z:

plt.figure(figsize=[6,2.5])

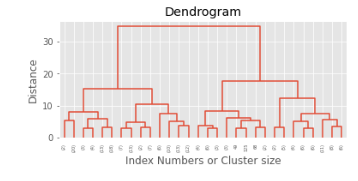
plt.title(‘Dendogram’)

plt.xlabel(‘Index Numbers or Cluster size’)

plt.ylabel(‘Distance’)

dendogram(Z, leaf\_rotation = 90, leaf\_font\_size = 5., color\_threshold = 0, truncate\_mode = ‘lastp’)

plt.show()



# Se define una distancia de corte en el eje horizontal en el dendograma para identificar clusters. Cada rama independiente del dendograma corresponde a un cluster. IE: si cortamos el árbol a la altura de distancia = 8, tenemos 8 clusters.

from scipy.cluster import hierarchy

plt.figure(figsize=[8,3.5])

plt.title(‘Dendogram’)

plt.xlabel(‘Data points’)

plt.ylabel(‘Distance’)

color\_palette=[‘r’, ’g’, ’y’, m‘, ‘c’]

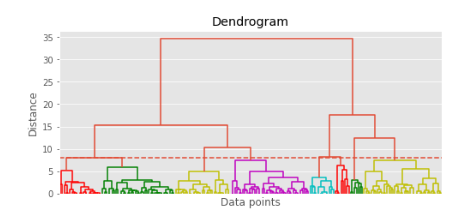
hierarchy.set\_link\_color\_palette(color\_palette)

dendrogram(Z, leaf\_rotation = 90, leaf\_font\_size = 5., color\_threshold = 8)

plt.hlines(8, 0, 2000, linestile=--’)

plt.xticks([])

plt.show()



# Con el método **fcluster** podemos ver las observaciones de cada cluster.

Labels = fcluster(Z, 8, criterion = ‘maxclust’)

# Vamos a ver el total de automóviles agrupados por cluster:

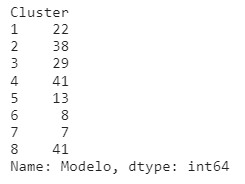
frames = [df[‘make’], df[’body\_style’], pd.DataFrame(labels, columns=[‘Cluster’])]

result = pd.concat(frames, axis = 1)

clusters = result.rename(columns = {‘make’:’Modelo’})

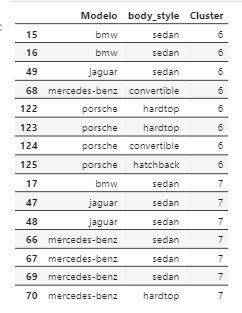
# clusters.loc[clusters[‘cluster’].isin([4])].sort\_values(‘Cluster’)

Clusters.groupby(‘Cluster’)[‘Modelo’].count()

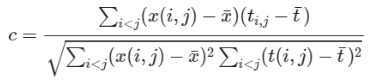


# Ahora vamos a ver el detalle de automóviles para dos clusters:

Clusters.loc[clusters[‘Cluster’].isin([6,7])].sort\_values(‘Cluster’)



El **Coeficiente Cofenético** compara las distancias originales entre las observaciones, con las distancias que tienen los clusters a los que pertenecen. El ideal es que esté cerca de 1, ya que esto indica que las dos distancias están muy correlacionadas:



x(i,j) es la distancia entre los puntos i y j.

t(i,j) es la distancia entre los clusters que contenían al punto i y al punto j al momento de unirse.

**En Python:**

# Vamos a calcular el coeficiente cofenético usando el linkage Ward:

c, dists = cophenet(Z, pdist(X\_scaled))

print(‘Cophenetic coefficient’, c)



# Vamos a comparer los distintos tipos de linkage y su respectivo coeficiente cofenético:

def plot\_dendrgram(X\_sc, method, ax):

Z = linkage(X\_sc, method)

c, dists = cophenet(Z, pdist(X\_sc))

ax.set\_title(method+’ c=’+str(np.around(c,2)))

ax.set\_xticklabels([])

dendrogram(Z, leaf\_rotation = 90, leaf\_font\_size = 5., color\_threshold = 0, truncate\_mode = ‘lastp’, ax = ax)

return

fig, axes = plt.subplots(2, 2, figsize = (8,7))

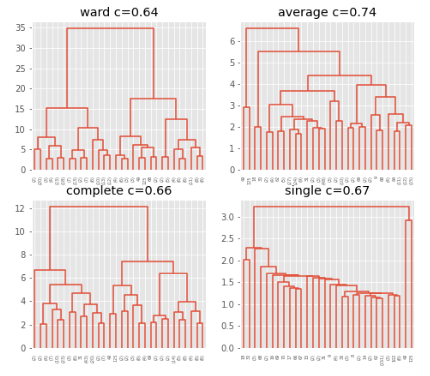
pot\_dendrogram(X\_scaled, ‘ward’, axes[0,0])

pot\_dendrogram(X\_scaled, ‘average’, axes[0,1])

pot\_dendrogram(X\_scaled, ‘complete’, axes[1,0])

pot\_dendrogram(X\_scaled, ‘single, axes[1,1])

plt.show()



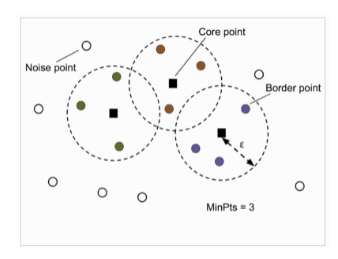
**Resumen – Cluster Jerárquico:**

* No necesita definición de K (Cantidad de Clusters) a priori.
* Ofrece una representación gráfica del proceso de generación de clusters: El dendrograma.
* Hay varios métodos para unir los clusters (Linkage)
* Es más costoso computacionalmente que K-Means.
* Al igual que K-Means, también puede fallar cuando los clusters no tienen geometría esférica.

**DBSCAN**: **Density Based Spatial Clustering of Applications with Noise**. Identifica clusters como regiones con alta densidad de puntos.

Este algoritmo tiene dos hiperparámetros:

1. **epsilon**: Es la máxima distancia entre dos puntos para considerarlos pertenecientes al mismo cluster (similar a cercanía).
2. **min\_points**: Mínimo número de puntos necesarios para formar un cluster(con esto evita formar clusters demasiado pequeños).



Se definen 3 tipos de puntos:

1. **Core:** Son puntos que tienen al mismo min\_points vecinos en un radio épsilon. Son **puntos de alta densidad**.
2. **Border**:Puntos que están dentro de un radio de un punto core, pero no son core. Son **las fronteras de los clusters**.
3. **Noise**: Puntos que no son ni core ni border. Son **outliers**, **ruido**. No pertenecen a ningún cluster.

Lo que hace DBSCAN es pararse en la idea de que un punto pertenece a un cluster si tiene cerca muchos puntos del cluster. El algoritmo básico consiste en:

1. Se selecciona un punto al azar. Se determina su vecindario usando un radio épsilon.
2. Si cuenta con al menos min\_points vecinos, entonces se lo define como core y se crea un cluster. Caso contrario, se lo define como noise.
3. En caso de haber sido definido como core, se define a sus vecinos dentro del radio epsilon como miembros del mismo cluster.
4. Si alguno de los puntos vecinos recién definidos habían sido definidos como core de otro cluster previamente, se reasignan al cluster recién definido, junto con sus vecinos.
5. Repite los pasos 1 a 3, tomando un nuevo punto no procesado.
6. Una vez que se visitaron todos los puntos, finaliza el proceso.

Un punto marcado como Noise puede ser revisado.

**En Python:**

# Vamos a crear un DataSet con el método **make\_moons**:

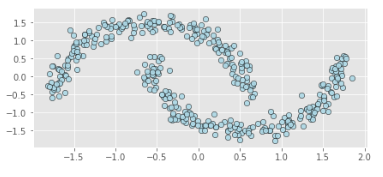
from sklearn import make\_moons

X, label = make\_moons(n\_samples = 400, noise = 0.7, random\_state = 19)

X = StandardScaler().fit\_transform(X)

fig, ax = plt.subplots(figsize=(7,3))

sctr = ax.scatter(X[:,0], X[:,1], c = ‘lightblue’, edgecolor = ‘black’, s = 40, alpha = 0.9, cmap = plt.cm.Set1)



# El método **DBSCAN** implementa el modelo. Con el método **fit\_predict** se pueden calcular los clusters sobre los datos:

from sklearn import DBSCAN

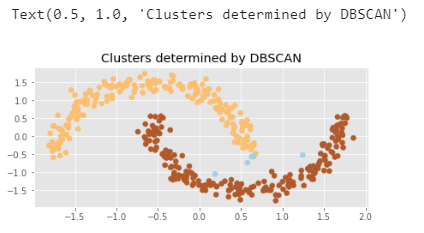
dbscan = DBSCAN(eps = 0.2, min\_samples = 5)

y\_pred = dbscan.fit\_predict(X)

plt.figure(figsize = (7,3))

plt.scatter(X[:,0], X[:,1], c=y\_pred, cmap = ‘Paired’)

plt.title(‘Clusters determined by DBSCAN’)



# Similar a como con los modelos K-Means y Hierarchy Cklustering, con DBSCAN podemos acceder al número de cluster de cada observación con la función **labels\_**. En DBSCAN, los outliers se identifican con -1.

dbscan.labels\_[dbscan.labels\_ == -1].size



Conociendo el dominio de los datos, **epsilon** es una distancia típica que tiene sentido físico, y **minPts** es el tamaño para los clusters.

En caso de no conocer bien el dominio de los datos, podemos recurrir a algunas heurísticas para determinar estos valores:

* **minPts:** Asignar este parámetro en función del número de dimensiones D del dataset: En general se elige nimPts = 2D, pero en casos de datasets muy grandes, con mucho ruido o con datos duplicados, podría adoptarse un valor mayor.
* **epsilon:** Podemos determinarla mirando un gráfico de distancias al k-ésimo vecino, con k = minPts – 1: Se calcula la distancia al k-ésimo vecino más cercano de cada punto y luego se grafican estas distancias de mayor a menor. Luego se puede definir epsilon como la distancia del codo (punto de quiebre de la curva).

**En Python:**

from scipy.spatial.distance import pdist, squareform

minPts = 5

k = minPts-1

D = squareform(pdist(X))

k\_distances = np.zeros(D.shape[0])

for i in range(D.shape[0]):

distances = np.sort(D[i])

k\_distances[i] = distances[k]

k\_distances = np.sort(k\_distances)

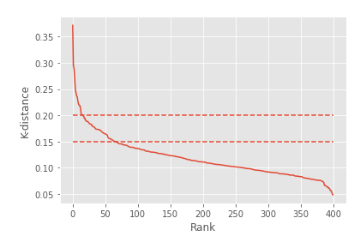
k\_distances = k\_distances[::-1]

plt.plot(k\_distances)

plt.xlabel(‘Rank’)

plt.ylabel(‘K-distance’)

plt.hlines([0.2, 0.15], 0, 400, linestyles =’dashed’)



Una elección de epsilon entre 0.15 y 0.2 parece estar bien.

**Resumen de DBSCAN:**

* Funciona muy bien en clusters con límites no lineales y de tamaños muy variados.
* No necesita que le definamos la cantidad de clusters (K) de antemano.
* Incorpora el concepto de **ruido**. Lo que lo hace robusto frente a **outliers** a diferencia de K-Means (no estoy seguro de Clustering Jerárquico).
* Puede que no trabaje bien si tenemos datasets con zonas de densidades muy distintas. Porque usaría un mismo epsilon y min\_points para comparar peras y caballos.
* Se puede complicar el cálculo de epsilon si no concemos bien los datos y su escala.