modelos_no_supervisados

March 2, 2023

[ ]:

[1]:
```
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from scipy.cluster.hierarchy import linkage
from scipy.cluster.hierarchy import dendrogram
from scipy.cluster.hierarchy import fcluster
import seaborn as sns
from sklearn.cluster import DBSCAN
```

[ ]:

1 1. Reducción de la Dimensionalidad

La reducción de la dimensionalidad es un proceso importante en el análisis de datos, ya que permite reducir la cantidad de variables o características de un conjunto de datos sin perder demasiada información. Uno de los métodos más comunes para reducir la dimensionalidad es el Análisis de Componentes Principales (PCA, por sus siglas en inglés).

El PCA es un método de reducción de dimensionalidad que se utiliza para transformar un conjunto de variables correlacionadas en un conjunto de variables no correlacionadas (también conocidas como "componentes principales") ordenadas en función de su varianza. La idea es que al reducir la cantidad de variables a través de la combinación de las variables existentes, se puede conservar la mayor parte de la información relevante del conjunto de datos.

[ ]:

1.1 1.1 PCA

[4]:
```
# Generamos un dataframe de prueba

np.random.seed(123)
X = np.random.rand(100, 5) * 10
X[:, 0] += X[:, 1] + X[:, 2]
X[:, 3] = 0.5 * X[:, 0] + 2 * X[:, 1] + X[:, 2] + np.random.randn(100) * 0.5
X[:, 4] = 0.5 * X[:, 0] + 2 * X[:, 1] + np.random.randn(100) * 0.5
df = pd.DataFrame(X, columns=['X1', 'X2', 'X3', 'X4', 'X5'])
df
```

[4]:

	X1	X2	X3	X4	X5
0	12.094600	2.861393	2.268515	13.793115	11.637364
1	20.887004	9.807642	6.848297	36.252501	30.523578
2	15.108000	7.290497	4.385722	26.516386	22.765405
3	10.959389	1.824917	1.754518	11.372453	9.798256
4	22.082881	8.494318	7.244553	34.399094	27.534939
..	…	…	…	…	…
95	7.251382	6.458225	0.386996	16.915676	15.844954
96	14.708828	6.484497	7.326012	27.805211	20.779752
97	10.324986	4.510883	2.871033	16.984267	13.547475
98	25.029508	9.882149	9.025565	41.680635	32.849381
99	27.827828	8.827130	9.194725	40.797719	31.174091

[100 rows x 5 columns]

[7]:
```
# Estandarizamos los datos, media 0 y sd = 1

scaler = StandardScaler()
X_std = scaler.fit_transform(df)
```

[8]:
```
# Creamos el PCA y ajustamos a 3 componentes

pca = PCA(n_components=3)
pca.fit(X_std)
```

[8]:
```
PCA(n_components=3)
```

[9]:
```
X_pca = pca.transform(X_std)
print('Varianza explicada por cada componente principal:', pca.
 ↪explained_variance_ratio_)
print('Contribución de cada variable a cada componente principal:', pca.
 ↪components_)
```

```
Varianza explicada por cada componente principal: [0.73543221 0.22636393
0.03715808]
Contribución de cada variable a cada componente principal: [[-0.46958359
```

```
-0.44707419 -0.24347883 -0.51933725 -0.50062241]
 [ 0.25368814 -0.45902651  0.81013231  0.03607351 -0.25946355]
 [ 0.79072883 -0.37268939 -0.4580066  -0.16026798 -0.01986614]]
```

[ ]:

1.2 1.2 PCA Kernel

El análisis de componentes principales con kernel (kernel PCA) es una técnica de reducción de dimensionalidad no lineal que puede ayudar a manejar conjuntos de datos complejos y no lineales. En lugar de utilizar una transformación lineal para reducir la dimensionalidad, kernel PCA utiliza una función kernel para mapear los datos a un espacio de características de mayor dimensión donde pueden ser más fáciles de separar.

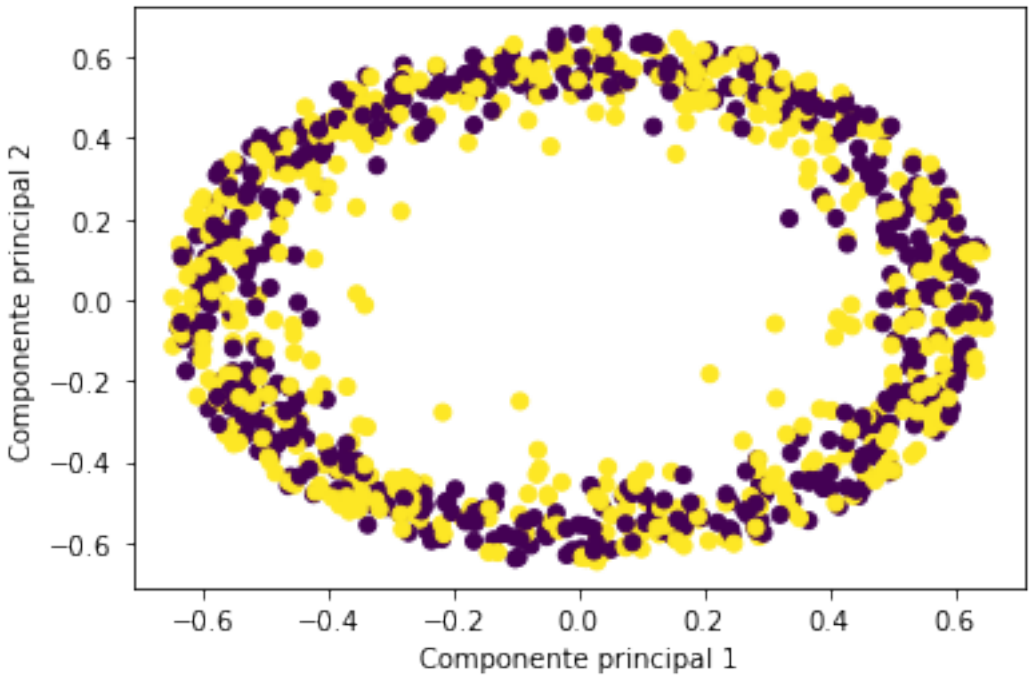
[10]:
```
# Generamos los datos

X, y = make_circles(n_samples=1000, noise=0.1, factor=0.5)
df = pd.DataFrame(X, columns=['X1', 'X2'])
df
```

[10]:
```
           X1        X2
0    0.836089 -0.594471
1   -0.814085 -0.550424
2    0.403170 -0.390890
3   -0.136538 -0.350572
4   -0.848754  0.193244
..        …       …
995  0.199614 -0.413619
996 -0.042153 -0.547669
997  0.520991  0.037389
998 -0.164085  0.589381
999  0.345862  0.575387

[1000 rows x 2 columns]
```

[11]:
```
# Estandarizamos los datos

scaler = StandardScaler()
X_std = scaler.fit_transform(df)
```

[12]:
```
# creamos el objeto Kernel y ajustamos por una radial de dos componentes

from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=2, kernel='rbf')
X_kpca = kpca.fit_transform(X_std)
```

[13]:
```
#Visualizamos los datos en el espacio de características de los CP utilizando␣
↪un diagrama de dispersión

plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y)
plt.xlabel('Componente principal 1')
plt.ylabel('Componente principal 2')
plt.show() # Vemos como la relación no es lineal
```



[ ]:
```

```

2 2. Clusters

El aprendizaje no supervisado es una rama del aprendizaje automático que se utiliza para descubrir patrones en los datos sin una etiqueta o respuesta conocida. Una técnica común en el aprendizaje no supervisado es el clustering, que se utiliza para agrupar los datos en grupos o clusters basados en su similitud.

Es importante destacar que el número de clusters (K) debe seleccionarse cuidadosamente. Una forma de hacerlo es utilizando el método del codo, que implica graficar la variabilidad explicada por el modelo para diferentes valores de K y seleccionar el valor de K en el que se produce una disminución significativa en la variabilidad explicada.

Existen varias formas de determinar el número óptimo de clusters en un conjunto de datos. Una forma común es utilizar el método del codo que mencioné anteriormente. El método del codo consiste en graficar la variabilidad explicada por el modelo para diferentes valores de K y seleccionar

el valor de K en el que se produce una disminución significativa en la variabilidad explicada.

A continuación te presento algunos de los métodos más populares:

1. K-Medias (K-Means): Este es uno de los métodos de clustering más utilizados. El algoritmo K-Means particiona los datos en K clusters, donde K es un número predefinido de clusters. El algoritmo funciona minimizando la suma de las distancias al cuadrado entre cada punto y su centroide.
2. Clustering Jerárquico: El Clustering Jerárquico es un método que crea una jerarquía de clusters en forma de un árbol (dendrograma). Puede ser aglomerativo, comenzando con cada punto como su propio cluster y fusionando clusters hasta que todos los puntos estén en el mismo cluster, o divisivo, comenzando con todos los puntos en un solo cluster y dividiéndolos en clusters más pequeños.
3. Clustering Basado en la Densidad: El Clustering Basado en la Densidad es un método que se enfoca en encontrar regiones de alta densidad de puntos. Un ejemplo popular es el algoritmo DBSCAN (Density-Based Spatial Clustering of Applications with Noise).
4. Clustering Espectral: El Clustering Espectral es un método que utiliza la matriz de similitud entre los puntos para crear clusters. Este método puede manejar bien conjuntos de datos de alta dimensionalidad y datos no lineales.
5. Clustering de Máxima Verosimilitud: El Clustering de Máxima Verosimilitud es un método que modela los datos utilizando una distribución de probabilidad y luego encuentra los parámetros que mejor explican los datos. Un ejemplo popular es el algoritmo Gaussian Mixture Model (GMM).

La elección del método de clustering más adecuado depende de las características específicas de los datos y del problema en cuestión. En general, no hay un método de clustering único que sea el mejor para todos los casos.

Dicho esto, algunos métodos de clustering son más flexibles que otros. Por ejemplo, el Clustering Jerárquico es muy flexible porque puede ser aglomerativo o divisivo, y puede manejar diferentes métricas de distancia. El Clustering Espectral también es bastante flexible, ya que puede manejar bien conjuntos de datos de alta dimensionalidad y datos no lineales.

En términos de popularidad, el método K-Means es uno de los más utilizados, debido a su simplicidad y eficiencia en grandes conjuntos de datos. Además, el K-Means se puede utilizar para una amplia variedad de aplicaciones, desde la segmentación de clientes hasta el análisis de imágenes.

2.1 2.1 K-Means

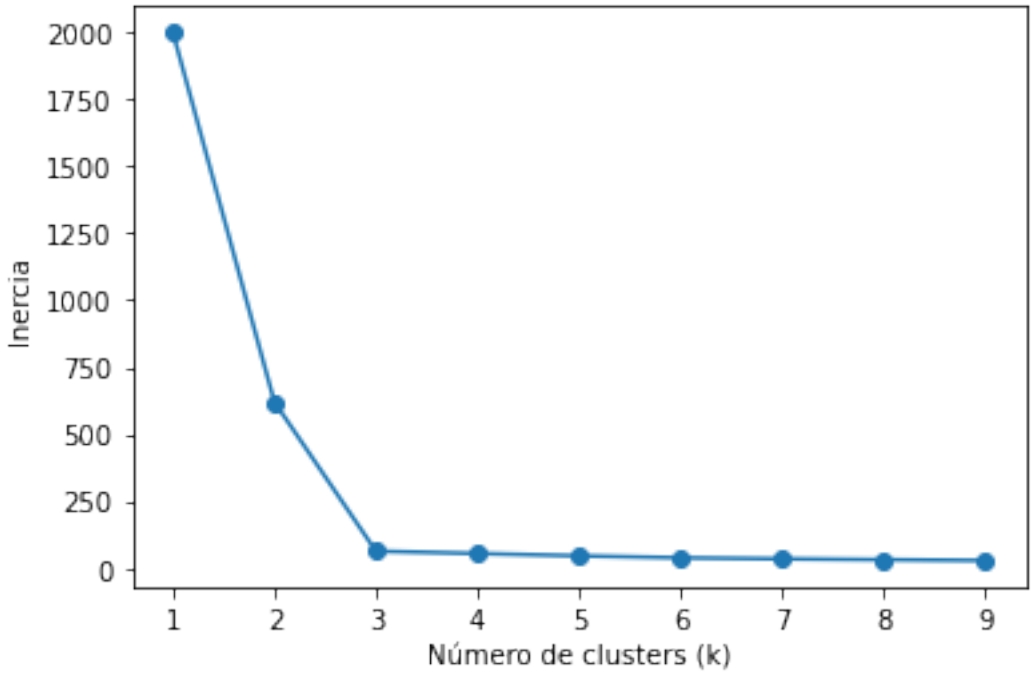
[16]:
```
# Generamos datos

X, y = make_blobs(n_samples=1000, centers=3, random_state=42)
df = pd.DataFrame(X, columns=['X1', 'X2'])
df
```

[16]:
```
           X1        X2
0   -6.596339 -7.139015
1   -6.137532 -6.580817
```

```
2    5.198206  2.049175
3   -2.968559  8.164442
4   -2.768789  7.511143
..        …       …
995 -4.818124 -5.671743
996 -1.885078  9.642632
997  3.690480  4.605552
998  4.030367  1.786198
999 -7.441795 -7.089331

[1000 rows x 2 columns]
```

[17]:
```
# Estandarizamos los datos

scaler = StandardScaler()
X_std = scaler.fit_transform(df)
```

[21]:
```
# Metodo del codo para especificar numero de clusters

# Calcular la inercia para diferentes valores de k
inertias = []
for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_std)
    inertias.append(kmeans.inertia_)

# Graficar la curva de codo
plt.plot(range(1, 10), inertias, marker='o')
plt.xlabel('Número de clusters (k)')
plt.ylabel('Inercia')
plt.xticks(np.arange(1, 10, 1))
plt.show()
# la elección correcta es 3. Despues de este la inercia baja casi que nada
```



[18]:
```
# Creamos el objeto K-means y ajustamos con tres clusters

kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X_std)
```

[18]:
```
KMeans(n_clusters=3, random_state=42)
```

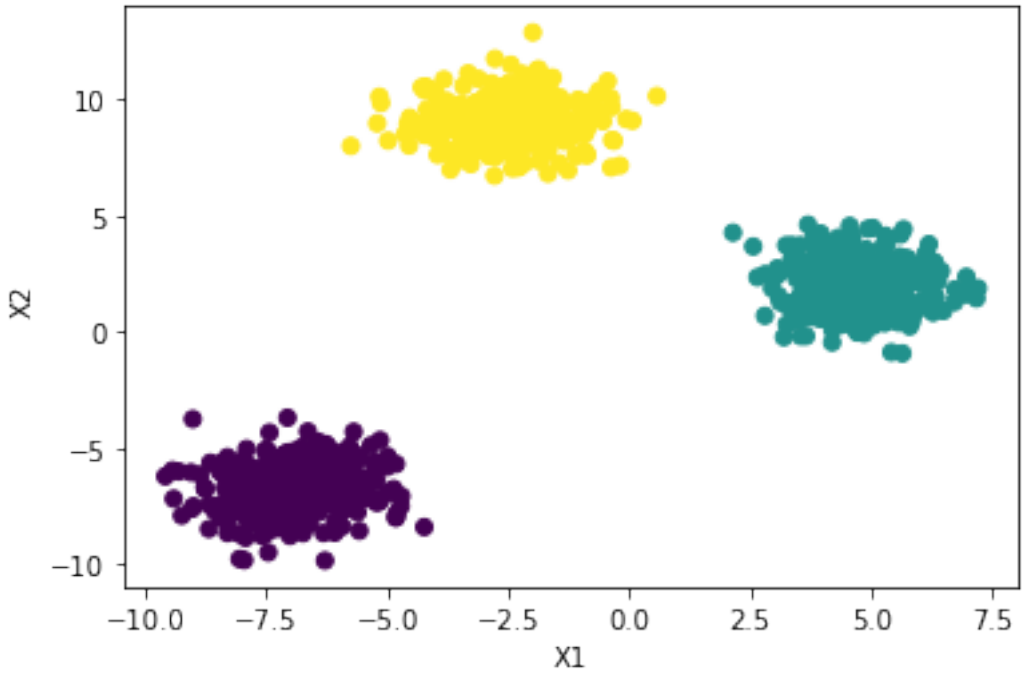
[19]:
```
# Añadimos labels del cluster a nuestros datos

df['cluster'] = kmeans.labels_
df
```

[19]:

	X1	X2	cluster
0	-6.596339	-7.139015	0
1	-6.137532	-6.580817	0
2	5.198206	2.049175	1
3	-2.968559	8.164442	2
4	-2.768789	7.511143	2
..	…	…	…
995	-4.818124	-5.671743	0
996	-1.885078	9.642632	2
997	3.690480	4.605552	1
998	4.030367	1.786198	1
999	-7.441795	-7.089331	0

```
[1000 rows x 3 columns]
```

[20]:
```
# Hacemos la visualización de como queda la agrupación

plt.scatter(df['X1'], df['X2'], c=df['cluster'])
plt.xlabel('X1')
plt.ylabel('X2')
plt.show()
```



[ ]:
```

```

2.2 2.2 Cluster jerárquico

[24]:
```
#Caragamos el dataframe

# Cargar los datos
data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪00236/seeds_dataset.txt',
                   sep='\t+',
                   engine = 'python',
                   header=None,
                   names=['area', 'perimeter', 'compactness', 'length',␣
 ↪'width', 'asymmetry', 'groove', 'class'])
data
```

[24]:

	area	perimeter	compactness	length	width	asymmetry	groove	class
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220	1
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825	1
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1
..	…	…	…	…	…	…	…	
205	12.19	13.20	0.8783	5.137	2.981	3.631	4.870	3
206	11.23	12.88	0.8511	5.140	2.795	4.325	5.003	3
207	13.20	13.66	0.8883	5.236	3.232	8.315	5.056	3
208	11.84	13.21	0.8521	5.175	2.836	3.598	5.044	3
209	12.30	13.34	0.8684	5.243	2.974	5.637	5.063	3

[210 rows x 8 columns]

[25]:
```
# Separar las características de la variable de clase

X = data.drop('class', axis=1)
X
```

[25]:

	area	perimeter	compactness	length	width	asymmetry	groove
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175
..	…	…	…	…	…	…	
205	12.19	13.20	0.8783	5.137	2.981	3.631	4.870
206	11.23	12.88	0.8511	5.140	2.795	4.325	5.003
207	13.20	13.66	0.8883	5.236	3.232	8.315	5.056
208	11.84	13.21	0.8521	5.175	2.836	3.598	5.044
209	12.30	13.34	0.8684	5.243	2.974	5.637	5.063

[210 rows x 7 columns]

[26]:
```
# Estandarizamos X

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

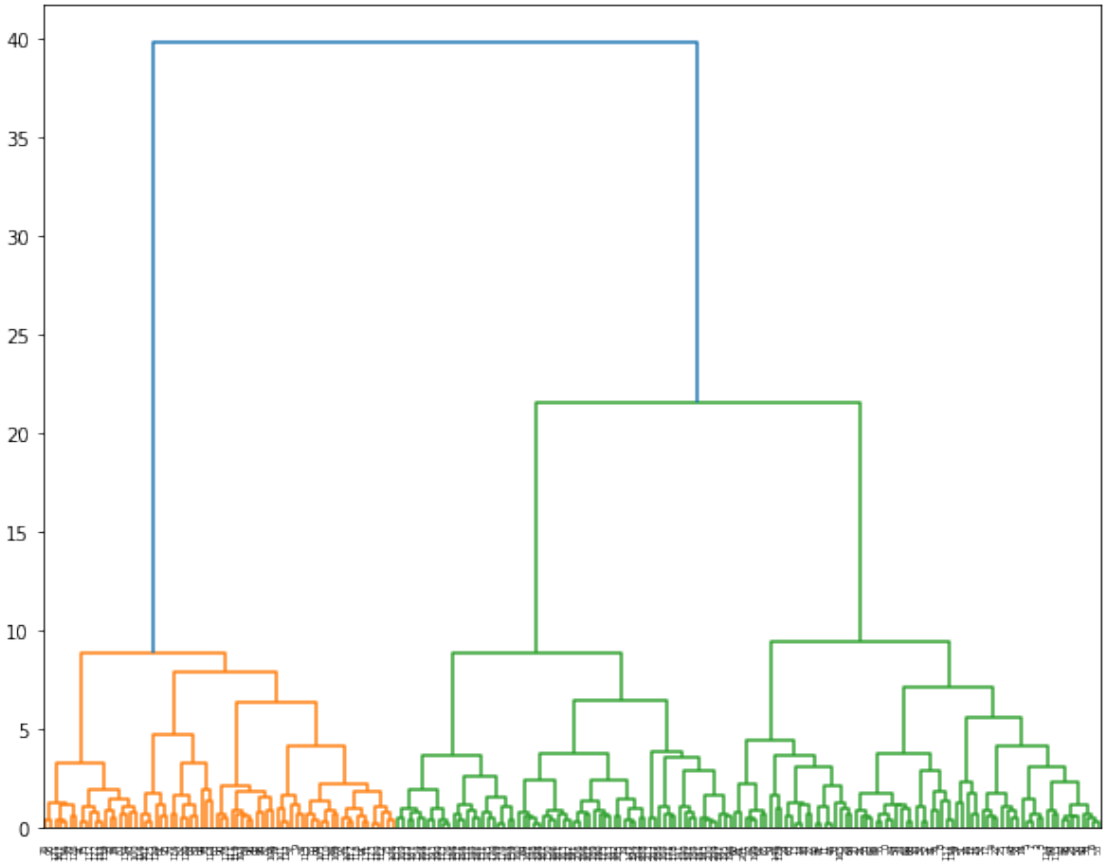
[27]:
```
# Calcular la matriz de enlaces. Contiene información sobre cómo se agrupan los␣
↪puntos en clusters

linkage_matrix = linkage(X_scaled, method='ward')
```

En el dendrograma, cada hoja representa un punto en el conjunto de datos y los clusters se forman al unir los puntos cercanos en el dendrograma. La altura de cada unión representa la distancia

entre los puntos. Los clusters se forman al cortar el dendrograma en una altura determinada.

Para determinar el número óptimo de clusters, podemos observar el dendrograma y buscar la altura en la que se forman los clusters más distintos. En este caso, parece que el dendrograma se puede cortar en 3 o 4 clusters:

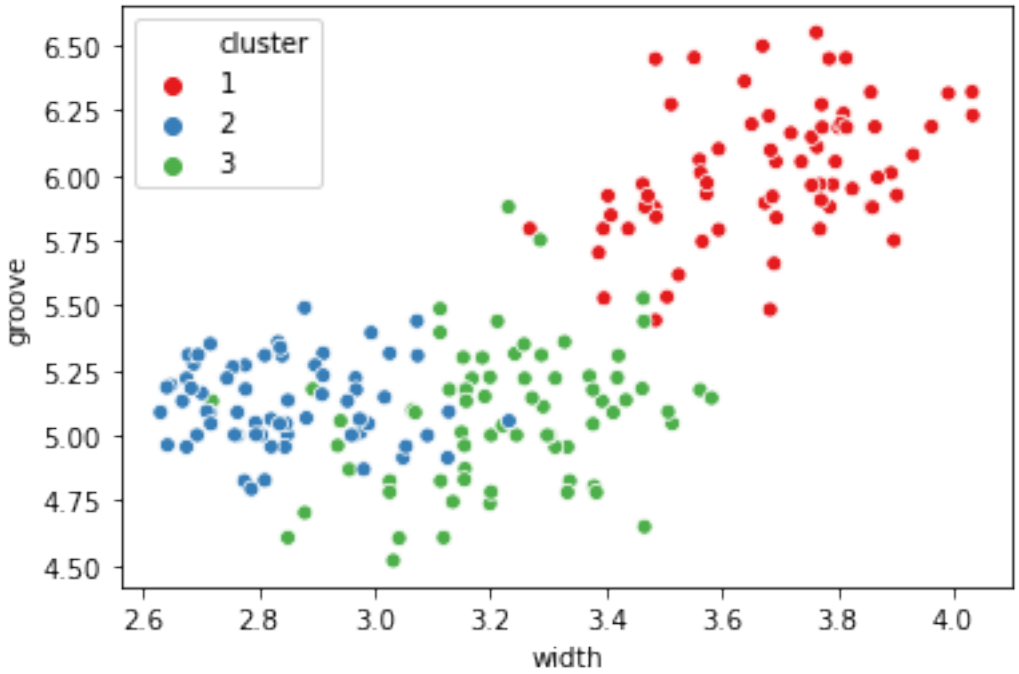
[28]:
```
# Visualizar el dendrograma

plt.figure(figsize=(10, 8))
dendrogram(linkage_matrix)
plt.show()
```



[30]:
```
# Asignar cada punto a un cluster específico, una vez visto cuantos cluster␣
↪óptimos

n_clusters = 3
clusters = fcluster(linkage_matrix, n_clusters, criterion='maxclust')
```

[31]:
```
# Añadimos las etiquetas y representamos la agrupación
```

```
data['cluster'] = clusters

sns.scatterplot(x='width', y='groove', hue='cluster', data=data, palette='Set1')
plt.show()
```



[ ]:

2.3 2.2 Cluster DBSCAN

Tanto K-means como DBSCAN son algoritmos de clustering, pero utilizan enfoques diferentes para identificar grupos de puntos similares. A continuación, se describen algunas de las principales diferencias entre ambos:

Método de agrupamiento: K-means es un método de agrupamiento particional, lo que significa que divide el conjunto de datos en un número fijo de clusters, mientras que DBSCAN es un método de agrupamiento basado en densidad, lo que significa que identifica regiones de alta densidad y agrupa puntos dentro de estas regiones.

Número de clusters: En K-means, el número de clusters se especifica antes de ejecutar el algoritmo, mientras que en DBSCAN, el número de clusters no se especifica y puede variar según la densidad de los datos.

Forma de los clusters: K-means asume que los clusters tienen una forma convexa y esférica, mientras que DBSCAN puede identificar clusters de cualquier forma.

Sensibilidad a los parámetros: K-means es menos sensible a los parámetros, ya que solo se especifica el número de clusters, mientras que DBSCAN es más sensible a los parámetros, ya que se deben

ajustar el radio eps y el número mínimo de puntos min_samples.

En cuanto a cuándo es preferible usar cada uno, depende de las características de los datos y los objetivos del análisis. En general, K-means puede ser más adecuado cuando el número de clusters es conocido o fácilmente estimable, los clusters tienen una forma convexa y esférica, y los datos no tienen una distribución de densidad clara. DBSCAN puede ser más adecuado cuando el número de clusters es desconocido o variable, los clusters tienen formas arbitrarias o no convexas, y los datos tienen una distribución de densidad clara. En cualquier caso, es recomendable probar diferentes algoritmos y ajustar los parámetros para encontrar la solución de clustering óptima.

[9]:
```
#Caragamos el dataframe

# Cargar los datos
data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪00236/seeds_dataset.txt',
                   sep='\t+',
                   engine = 'python',
                   header=None,
                   names=['area', 'perimeter', 'compactness', 'length',␣
 ↪'width', 'asymmetry', 'groove', 'class'])
data
```

[9]:

	area	perimeter	compactness	length	width	asymmetry	groove	class
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220	1
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825	1
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1
..	…	…	…	…	…	…	…	…
205	12.19	13.20	0.8783	5.137	2.981	3.631	4.870	3
206	11.23	12.88	0.8511	5.140	2.795	4.325	5.003	3
207	13.20	13.66	0.8883	5.236	3.232	8.315	5.056	3
208	11.84	13.21	0.8521	5.175	2.836	3.598	5.044	3
209	12.30	13.34	0.8684	5.243	2.974	5.637	5.063	3

[210 rows x 8 columns]

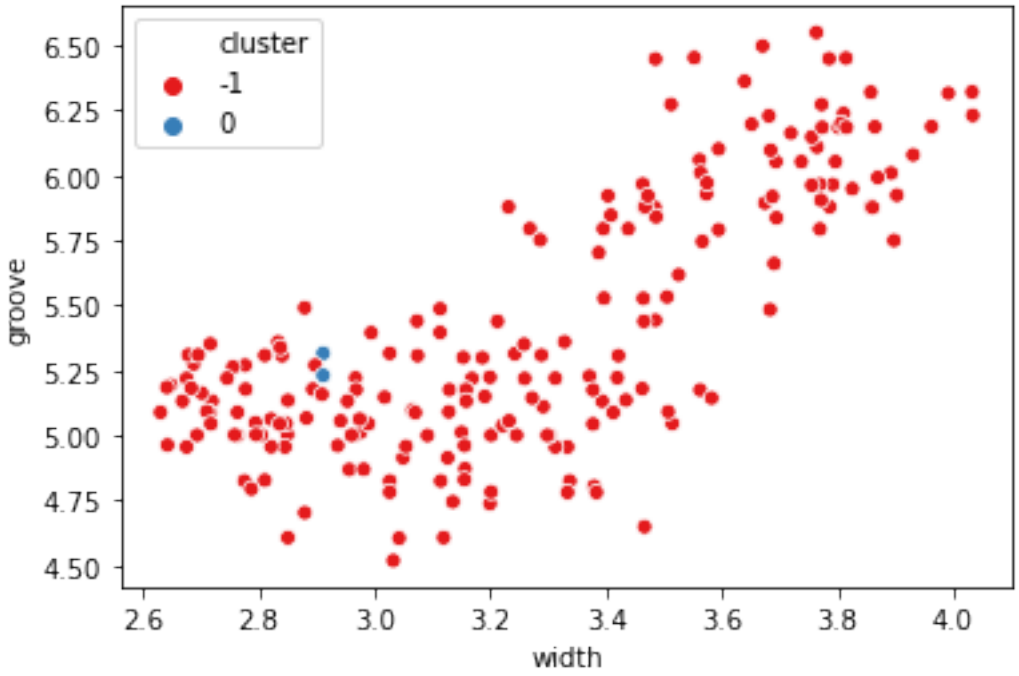
[10]:
```
# Separamos los regresores y estandarizamos

X = data.drop('class', axis=1)

# Normalizar las características
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

El parámetro eps controla la distancia máxima entre los puntos para que se consideren parte del mismo cluster, mientras que min_samples controla el número mínimo de puntos que deben estar dentro de la distancia eps para que se considere un cluster válido. Estos parámetros deben ajustarse

de manera adecuada según el conjunto de datos y los requisitos del problema.

[11]:
```
# Crear y ajustar el modelo DBSCAN

dbscan = DBSCAN(eps=0.5, min_samples=5)
clusters = dbscan.fit_predict(X_scaled)
```

[12]:
```
# Añadimos etiquetas y representamos

# Agregar las etiquetas de los clusters al dataframe original
data['cluster'] = clusters

# Visualizar los clusters en un scatterplot
import seaborn as sns
sns.scatterplot(x='width', y='groove', hue='cluster', data=data, palette='Set1')
plt.show()
# Como vemos solo ha identificado un grupo, el 0. Los -1 son puntos sin asignar.
```



[ ]: