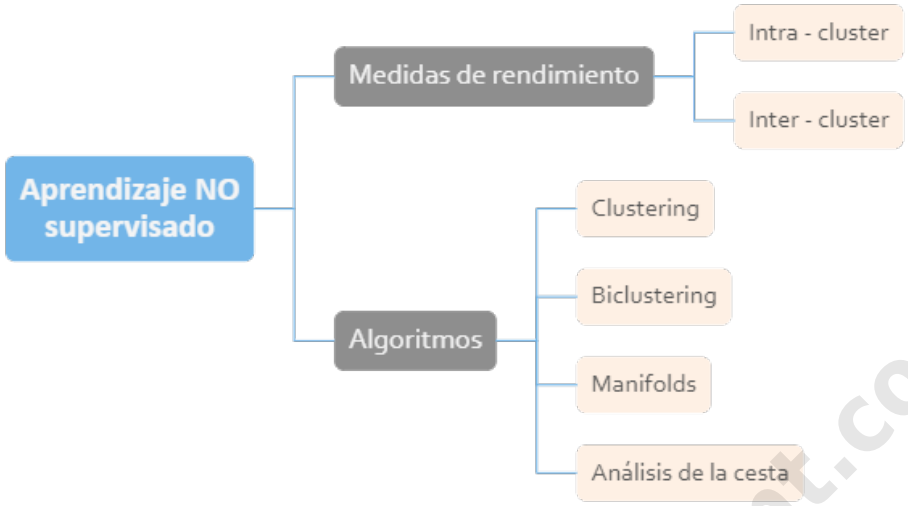
Introducción

En el **aprendizaje no supervisado**, deseamos conocer la estructura inherente de nuestros datos sin utilizar etiquetas proporcionadas explícitamente. Las tareas más comunes dentro del aprendizaje no supervisado son la agrupación, el aprendizaje de representación y la estimación de densidad.

Objetivos

- Conocer el aprendizaje no supervisado.
- Estudiar las aplicaciones del aprendizaje no supervisado.
- Comparar las medidas de rendimiento del aprendizaje no supervisado.
- Aprender a desarrollar modelos de clustering.
- Estudiar algoritmos biclustering.
- Aprender a trabajar con Manifolds.
- Valorar la implementación de análisis de la cesta.

Mapa Conceptual



Definición y aplicaciones.

El **aprendizaje no supervisado**, es el método que entrena a las máquinas para usar datos que no están clasificados ni etiquetados. Significa que no se pueden proporcionar datos de entrenamiento y la máquina está hecha para aprender por sí misma. La máquina debe ser capaz de clasificar los datos sin ninguna información previa sobre los datos.

La idea es exponer las máquinas a grandes volúmenes de datos variables y permitirles aprender de esos datos para proporcionar información que antes era desconocida e identificar patrones ocultos. Como tal, no hay necesariamente resultados definidos de algoritmos de aprendizaje no supervisados. Más bien, determina qué es diferente o interesante del conjunto de datos dado.

La máquina necesita ser programada para aprender por sí misma. La computadora necesita comprender y proporcionar información a partir de datos estructurados y no estructurados.

Las **categorías del aprendizaje automático** no supervisado son las siguientes:

- Clustering (agrupación en clústeres): es uno de los métodos de aprendizaje no supervisado más comunes. El método de agrupación en clústeres implica organizar los datos no etiquetados en grupos similares llamados clústeres. Por lo tanto, un clúster es una colección de elementos de datos similares. El objetivo principal aquí es encontrar similitudes en los puntos de datos y agrupar puntos de datos similares en un clúster.

- La detección de anomalías: es el método para identificar elementos raros, eventos u observaciones que difieren significativamente de la mayoría de los datos. Generalmente buscamos anomalías o valores atípicos en los datos porque son sospechosos. La detección de anomalías se utiliza a menudo en el fraude bancario y la detección de errores médicos.

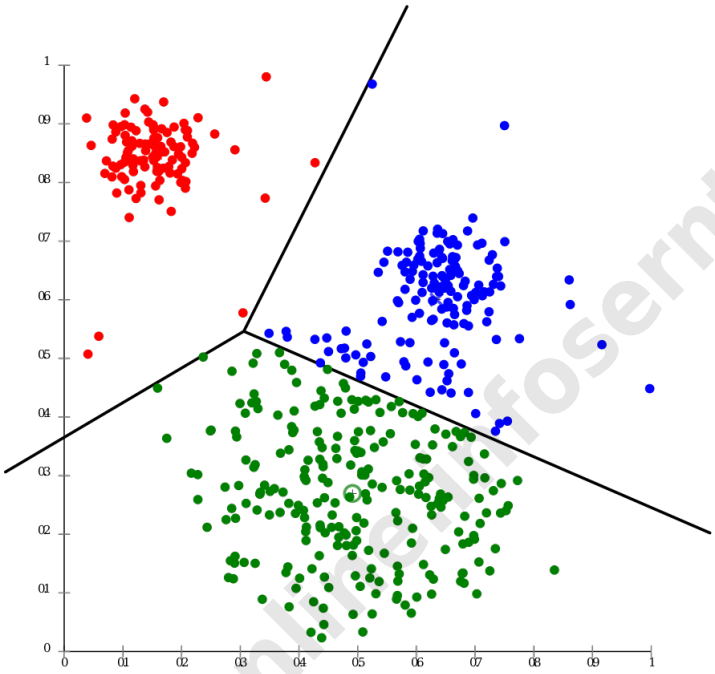
Algunas aplicaciones prácticas de los algoritmos de aprendizaje no supervisado incluyen:

- Detección de fraudes.
- Detección de malware.
- Identificación de errores humanos durante la entrada de datos.
- Realización de análisis precisos de la cesta, etc.

A continuación, vemos brevemente algunos algoritmos de aprendizaje no supervisado.

Algoritmos de clustering (agrupación)

Clustering es la tarea de agrupar un conjunto de objetos de modo que los objetos en el mismo grupo (cluster) sean más similares entre sí que con los de otros grupos.



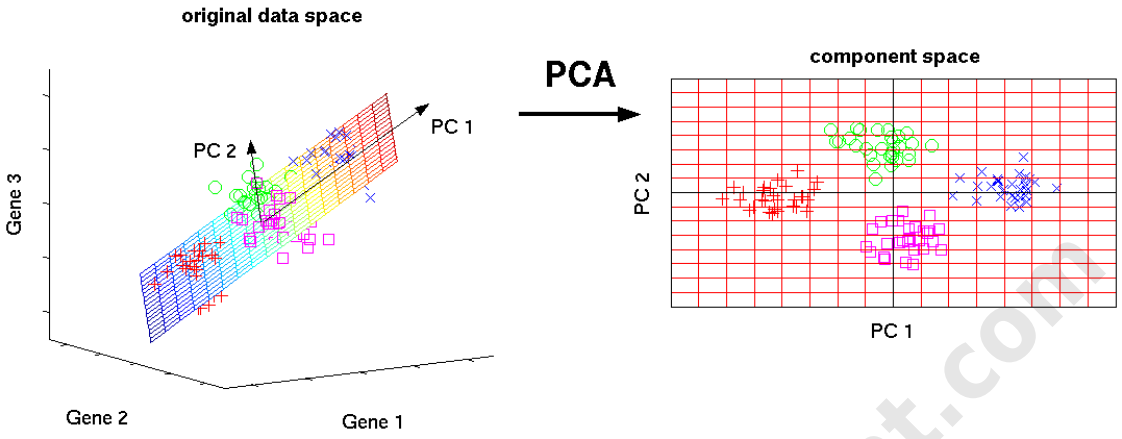
Cada algoritmo de clustering es diferente, veamos algunos ejemplos:

- Algoritmos basados en centroides.
- Algoritmos basados en conectividad.
- Algoritmos basados en densidad.
- Probabilístico.
- Reducción de dimensionalidad.
- Redes neuronales / aprendizaje profundo.

Análisis de componentes principales (PCA)

Es un procedimiento estadístico que utiliza una transformación ortogonal para convertir un conjunto

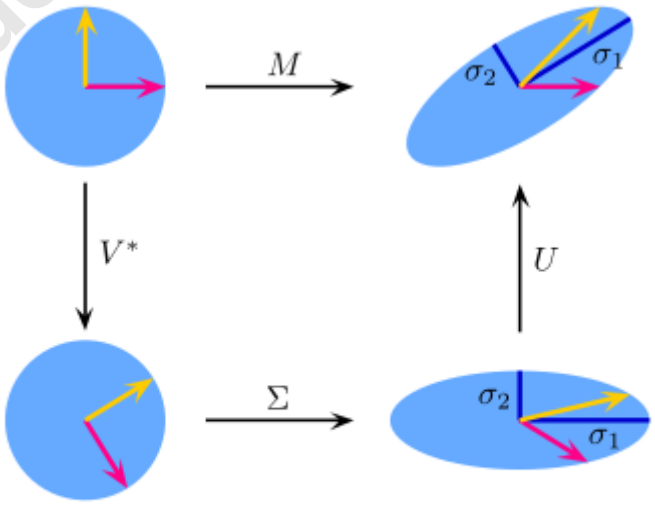
de observaciones de variables posiblemente correlacionadas en un conjunto de valores de variables no correlacionadas linealmente llamadas componentes principales.



Algunas de las aplicaciones de PCA incluyen compresión y simplificación de datos para facilitar el aprendizaje y la visualización. Hay que tener en cuenta que el conocimiento del dominio es muy importante al elegir si nos interesa aplica PCA o no. No es adecuado en casos donde los datos son ruidosos, porque todos los componentes de PCA tienen una variación bastante alta.

Descomposición de valor singular (SVD)

En álgebra lineal, SVD es una factorización de una matriz compleja real. Para una matriz m * n M dada, existe una descomposición tal que M = UΣV, donde U y V son matrices unitarias y Σ es una matriz diagonal.

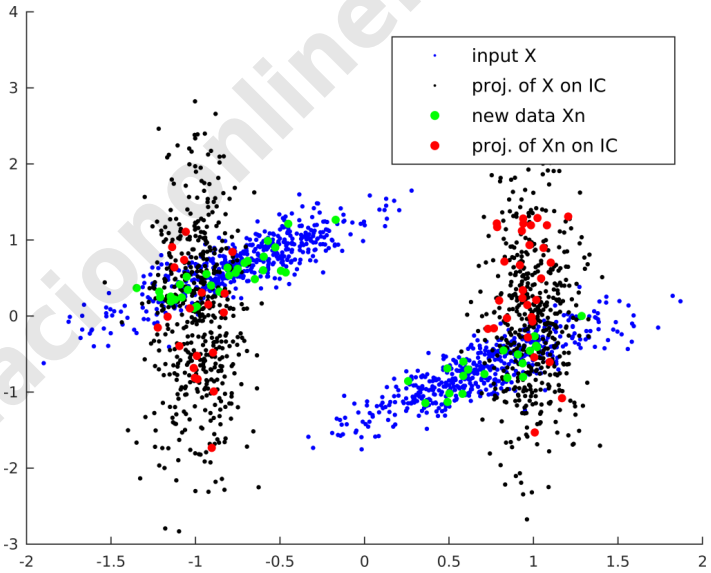


$$M = U \cdot \Sigma \cdot V^*$$

PCA es en realidad una aplicación simple de SVD. En la visión por computadora, los algoritmos de reconocimiento de la primera cara utilizaban PCA y SVD para representar caras como una combinación lineal de "caras propias", Después se llevaba a cabo una reducción de dimensiones y se hacían coincidir las caras con las identidades mediante métodos simples. Aunque los métodos modernos son mucho más sofisticados, muchos aún dependen de técnicas similares.

Análisis de componentes independientes (ICA)

Es una técnica estadística para revelar factores ocultos que subyacen en conjuntos de variables aleatorias, medidas o señales. ICA define un modelo generativo para los datos multivariable observados, que generalmente se proporciona como una gran base de datos de muestras. En el modelo, se supone que las variables de datos son mezclas lineales de algunas variables latentes desconocidas, el sistema de mezcla también es desconocido. Se supone que las variables latentes no son gaussianas y son mutuamente independientes, y se denominan componentes independientes de los datos observados.



ICA está relacionado con **PCA**, pero es una **técnica** mucho más poderosa que es capaz de encontrar los factores subyacentes de las fuentes cuando los métodos clásicos fallan. Sus aplicaciones incluyen imágenes digitales, bases de datos de documentos, indicadores económicos y mediciones psicométricas.

Clustering es la tarea de agrupar un conjunto de objetos de modo que los objetos

en el mismo grupo (cluster) sean más similares entre sí que con los de otros grupos.

Verdadero.
Falso.

Medidas de rendimiento.

Las **técnicas comunes de aprendizaje** no supervisado incluyen **clustering**, detección de anomalías y redes neuronales. Cada técnica requiere un método diferente para evaluar el rendimiento.

La agrupación en clústeres es la tarea de agrupar un conjunto de objetos de tal manera que los objetos del mismo clúster se parezcan más entre sí que a los objetos de otros clústeres. Varios algoritmos son capaces de agrupar, como k-means y jerárquicas, según las definiciones de un clúster y cómo encontrar uno.

Supongamos que necesitamos agrupar a los clientes bancarios en grupos en función de la cantidad y magnitud del riesgo que plantean. Después de que el algoritmo de agrupación en clústeres haya agrupado a los clientes en clústeres distintos, debemos evaluar lo bien que se formaron esos clústeres.

La falta de etiquetas en los datos de entrenamiento de un modelo de aprendizaje no supervisado hace que la evaluación sea problemática porque no hay nada con lo que los resultados del modelo puedan compararse de manera significativa. Si tuviéramos que agrupar manualmente a estos clientes, podríamos comparar nuestras agrupaciones manuales con las del algoritmo, pero a menudo esta no es una opción debido a limitaciones de tiempo o falta de mano de obra, por lo que necesitamos una forma más eficiente de determinar lo bien que funcionó el algoritmo.

Una forma sería determinar:

- Cómo de cerca está cada cliente dentro de cada clúster de cada otro cliente en su mismo clúster (la distancia "intra-cluster")
- Cómo de cerca está cada grupo de clientes de otros clústeres (la distancia "inter-cluster")

Y finalmente comparar las dos distancias.

Los modelos que producen distancias **intra-cluster** relativamente pequeñas y distancias **inter-cluster** relativamente grandes se evalúan favorablemente porque parecen estar haciendo un buen trabajo de agrupación de clientes con características discretas.

Clustering. Tipos

En el **aprendizaje supervisado**, hemos etiquetado los datos, por lo que tiene salidas que sabe con certeza que son los valores correctos para sus entradas. Es como conocer los precios de los automóviles en función de características como la marca, el modelo, el estilo, la transmisión y otros atributos.

Con el aprendizaje semi-supervisado, tenemos un gran **conjunto de datos** donde algunos de los datos están etiquetados pero la mayoría no.

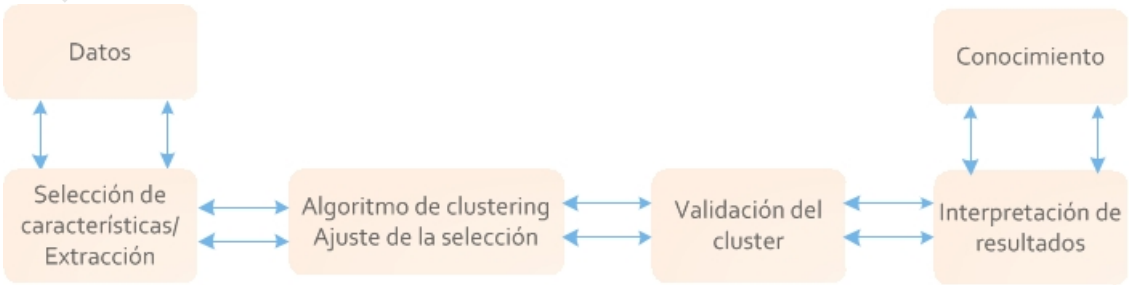
Esto cubre una gran cantidad de datos del mundo real porque puede resultar costoso conseguir que un experto etiquete cada punto de datos. Es posible solucionar este problema utilizando una combinación de aprendizaje supervisado y no supervisado.

El aprendizaje no supervisado significa que tenemos un conjunto de datos que está completamente sin etiquetar. No sabe si hay patrones ocultos en los datos, por lo que deja que el algoritmo encuentre todo lo que pueda.

Ahí es donde entran en juego los algoritmos de agrupación en clústeres. Es uno de los métodos que podemos utilizar en un problema de aprendizaje sin supervisión.

El **clustering (agrupación en clústeres)** es una tarea de aprendizaje automático no supervisada. También se denomina análisis de clusters, debido a la forma en que funciona este método.

El uso de un algoritmo de agrupación en clústeres significa que le dará al algoritmo una gran cantidad de datos de entrada sin etiquetas y le permitirá encontrar cualquier agrupación que pueda en los datos.



Proceso de algoritmo de clustering

Esas agrupaciones se denominan clusters. Un clúster es un grupo de **puntos de datos** que son similares entre sí en función de su relación con los puntos de datos circundantes. La agrupación en clústeres se utiliza para cosas como la ingeniería de características o el descubrimiento de patrones.

Cuando se comienza con datos de los que no sabemos nada, la agrupación en clústeres puede ser un buen lugar para obtener información.

Tipos de algoritmos de clustering

Existen diferentes tipos de algoritmos de agrupamiento que manejan todo tipo de datos únicos.

Basado en densidad

En la agrupación basada en densidad, los datos se agrupan por áreas de altas concentraciones de puntos de datos rodeadas por áreas de bajas concentraciones de puntos de datos. Básicamente, el algoritmo encuentra los lugares que son densos con puntos de datos y llama a esos clústeres.

Lo mejor de esto es que los clústeres pueden tener cualquier forma. No está limitado a las condiciones esperadas.

Los algoritmos de agrupación en clústeres de este tipo no intentan asignar valores atípicos a los clústeres, por lo que se ignoran.

Basado en distribución

Con un enfoque de agrupamiento basado en distribución, todos los puntos de datos se consideran partes de un clúster en función de la probabilidad de que pertenezcan a un grupo determinado.

Hay un punto central y, a medida que aumenta la distancia entre un punto de datos y el centro, la probabilidad de que forme parte de ese grupo disminuye.

Si no estamos seguros de cómo podría ser la distribución de sus datos, deberíamos considerar un tipo diferente de algoritmo.

Basado en centroides

La agrupación en clústeres basada en centroides es la que probablemente es más popular. Es un

poco sensible a los parámetros iniciales, pero es rápido y eficiente.

Estos tipos de algoritmos separan los puntos de datos en función de múltiples centroides en los datos. Cada punto de datos se asigna a un grupo en función de su distancia al cuadrado del centroide. Este es el tipo de agrupación más utilizado.

De base jerárquica

La agrupación en clústeres jerárquica se utiliza normalmente en datos jerárquicos, como se obtendría de una base de datos de empresa o taxonomías. Construye un árbol de grupos para que todo esté organizado de arriba hacia abajo.

Esto es más restrictivo que los otros tipos de clústeres, pero es perfecto para tipos específicos de conjuntos de datos.

Cuándo usar la agrupación en clústeres

Cuando tenemos un conjunto de datos sin etiquetar, es muy probable que utilicemos algún tipo de algoritmo de aprendizaje sin supervisión.

Hay muchas técnicas diferentes de **aprendizaje no supervisado**, como redes neuronales, aprendizaje por refuerzo y agrupación. El tipo específico de algoritmo que deseamos utilizar dependerá de cómo se vean nuestros datos.

Es posible que deseemos utilizar la agrupación en clústeres cuando intentemos realizar una detección de anomalías para intentar encontrar valores atípicos en los datos. Ayuda a encontrar esos grupos de clústeres y muestra los límites que determinarían si un punto de datos es un valor atípico o no.

Si no estamos seguros de qué funciones usar para un modelo de aprendizaje automático, la agrupación en clústeres descubre patrones que podemos usar para descubrir qué se destaca en los datos.

La agrupación en clústeres es especialmente útil para explorar datos de los que no sabe nada. Puede llevar algún tiempo averiguar qué tipo de algoritmo de agrupación en clústeres funciona mejor, pero cuando lo hagamos, obtendremos información muy valiosa sobre los datos. Es posible que

encontremos conexiones en las que nunca hubiéramos pensado.

Algunas aplicaciones del mundo real de la agrupación en clústeres incluyen la **detección de fraudes en seguros**, la categorización de libros en una biblioteca y la segmentación de clientes en marketing. También se puede utilizar en problemas más grandes, como análisis de terremotos o planificación urbana.

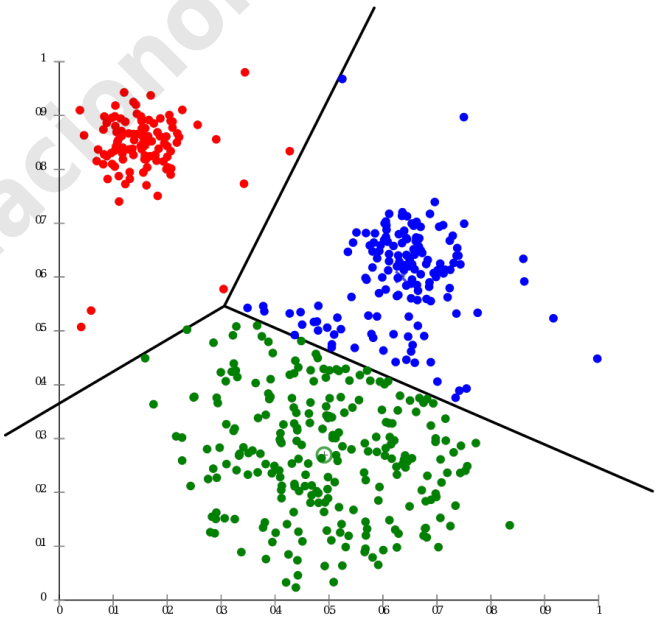
Algoritmos de clustering

Es posible implementar estos algoritmos en un conjunto de datos de ejemplo de la biblioteca sklearn en Python.

Usaremos el conjunto de datos make_classification de la biblioteca sklearn para demostrar cómo los diferentes algoritmos de agrupamiento no son adecuados para todos los problemas de agrupamiento.

Algoritmo de agrupación en clústeres de K-means

La agrupación en clústeres de K-means es el algoritmo de agrupación en clúster más utilizado. Es un algoritmo basado en centroide y el algoritmo de aprendizaje no supervisado más simple.



Este algoritmo intenta minimizar la varianza de los puntos de datos dentro de un grupo. También es

la forma en que la mayoría de las personas conocen el aprendizaje automático sin supervisión.

K-means funciona mejor en conjuntos de datos más pequeños porque itera sobre todos los puntos de datos. Eso significa que llevará más tiempo clasificar los puntos de datos si hay una gran cantidad de datos en el conjunto de datos.

Debido a la forma en que k-means agrupa los puntos de datos, no se escala bien.

Implementación:

El **método K-Means** del módulo **sklearn.cluster** hace que la implementación del algoritmo K-Means sea más fácil. Partimos de un conjunto de 10 puntos de datos.

Object	X_value	Y_value
Object 1	1.005079	4.594642
Object 2	1.128478	4.328122
Object 3	2.117881	0.726845
Object 4	0.955626	4.385907
Object 5	-1.35402	2.769449
Object 6	-1.07295	2.627009
Object 7	-2.0375	3.048606
Object 8	2.354083	0.856632
Object 9	2.14404	0.964399
Object 10	1.166288	4.273516

Utilizar scikit-learn para K-Means clustering

from sklearn.cluster import KMeans

Especificar el número clusters (3) y fit datos X

kmeans = KMeans(n_clusters=3, random_state=0).fit(X)

Especificamos que el número de clústeres deseados es 3 (el valor de K). Después de eso, damos los puntos de datos como entradas al modelo K-Means y entrenamos el modelo.

Extraigamos ahora los centroides de clúster y las etiquetas de clúster de la variable K-Means.

Obtener los centroides de los clústeres

```
print(kmeans.cluster_centers_)

# Obtener las etiquetas de los clústeres

print(kmeans.labels_)
```

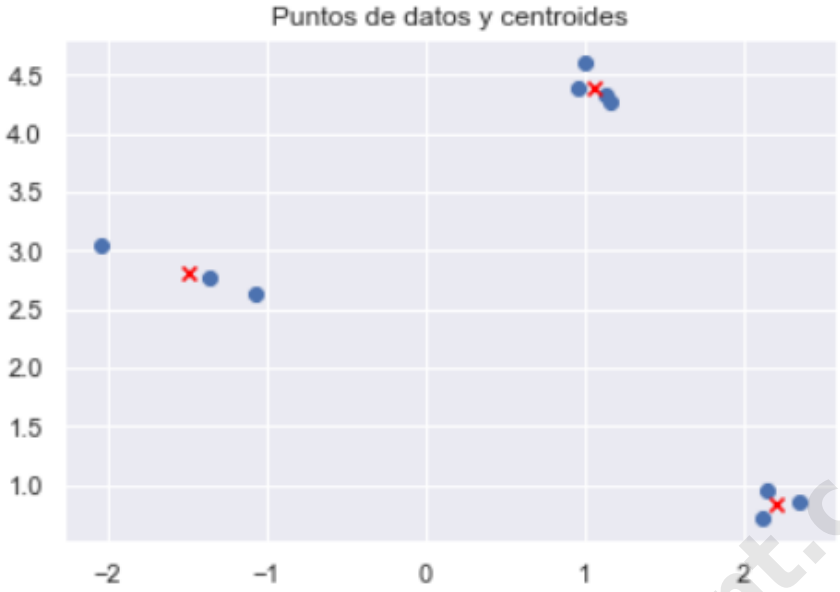
Produce el siguiente resultado:

```
[[ 1.06386762 4.39554682]

[ 2.20533486 0.84929168]

[-1.48815728 2.81502145]]

[0 0 1 0 2 2 2 1 1 0]
```

Vamos a trazar los **centroides del clúster** con respecto a los puntos de datos, para tener una representación gráfica.

```
# Dibujar los centros de los clústeres y los puntos de datos en 2D

plt.scatter(X[:, 0], X[:, -1])

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], c='red', marker='x')

plt.title('Puntos de datos y centroides')

plt.show()
```

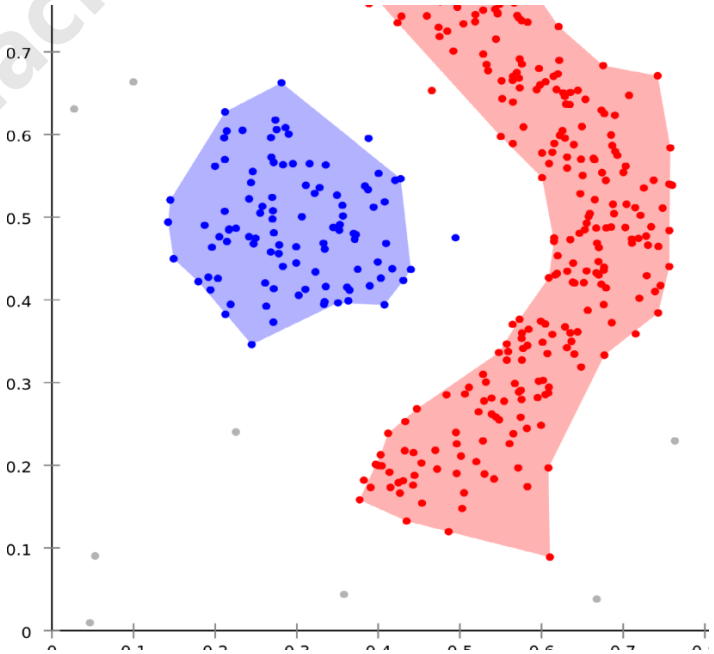
Obtenemos un gráfico como el siguiente:



Algoritmo de agrupación en clústeres DBSCAN

DBSCAN significa agrupación espacial basada en densidad de aplicaciones con ruido. Es un algoritmo de agrupamiento basado en densidad, a diferencia de k-means.

Este es un buen **algoritmo** para encontrar esquemas en un **conjunto de datos**. Encuentra grupos de forma arbitraria en función de la densidad de puntos de datos en diferentes regiones. Separa las regiones por áreas de baja densidad para que pueda detectar valores atípicos entre los grupos de alta densidad.



Este algoritmo es mejor que k-means cuando se trata de trabajar con datos de formas extrañas.

DBSCAN utiliza dos parámetros para determinar cómo se definen los grupos: minPts (el número mínimo de puntos de datos que deben agruparse para que un área se considere de alta densidad) y eps (la distancia utilizada para determinar si un punto de datos está en el misma área que otros puntos de datos).

La elección de los parámetros iniciales correctos es fundamental para que este algoritmo funcione.

Implementación:

Importaremos el conjunto de datos basado en blobs en sklearn. También importaremos clusterDBSCAN.

NumPy (as) se utilizará para el procesamiento de números y **Matplotlib PyPlot ()** para visualizar el conjunto de datos generado después de la agrupación en clústeres.

```
from sklearn.datasets import make_blobs

from sklearn.cluster import DBSCAN

import numpy as np

import matplotlib.pyplot as plt
```

Para generar el conjunto de datos, haremos dos cosas: especificar algunas opciones de configuración y usarlas en la llamada. También utilizaremos make_blobsepsilonmin_samples.

```
# Configuración

num_samples_total = 1000

cluster_centers = [(3,3), (7,7)]

num_classes = len(cluster_centers)

epsilon = 1.0

min_samples = 13
```

```
# Generar datos

X, y = make_blobs(n_samples = num_samples_total, centers = cluster_centers, n_features =
num_classes, center_box=(0, 1), cluster_std = 0.5)
```

Los **datos** serán ligeramente diferentes ya que se generan al azar.

Para poder repetir el proceso, puede ser aconsejable guardar los datos solo una vez después de ejecutar el script: al quitar los comentarios de la línea (np.save…), siempre cargará los mismos datos del archivo. Sin embargo, esto no es necesario.

```
# Grabar datos si se considera necesario

np.save('./clusters.npy', X)

X = np.load('./clusters.npy')
```

Ahora podemos inicializar DBScan y calcular los clústeres.

- Inicializamos con nuestros valores con . DBSCANepsilonmin_samples.
- Luego ajustamos inmediatamente los datos a DBSCAN, lo que significa que comenzará la agrupación en clústeres.
- Cargamos las etiquetas generadas (es decir, índices de clúster) una vez finalizado el agrupamiento.

```
# Computar DBSCAN

db = DBSCAN(eps=epsilon, min_samples=min_samples).fit(X)

labels = db.labels_
```

En nuestro caso, la impresión del número de clústeres y el número de muestras ruidosas produce 2 clústeres con 0 muestras ruidosas debido a nuestra selección de ε=1,0; minPts=13. En su caso, los resultados probablemente serán diferentes.

Trabajar con distintos **valores de épsilon** (es decir, hacer el círculo más grande) o un número mínimo de muestras (dependiendo de la densidad de sus grupos) producirá otros resultados.

Relaciona los elementos de la columna Derecha con la columna Izquierda

Algoritmo de clustering basado en densidad 1	Los datos se agrupan por áreas de altas concentraciones de puntos de datos rodeadas por áreas de bajas concentraciones de puntos de datos.
Algoritmo de clustering basado en centroides 2	Es la que probablemente es más popular. Es un poco sensible a los parámetros iniciales, pero es rápido y eficiente.


```
no_clusters = len(np.unique(labels) )

no_noise = np.sum(np.array(labels) == -1, axis=0)

print('Estimación no. clusters: %d' % no_clusters)

print('Estimación no. noise points: %d' % no_noise)
```

Resultado:

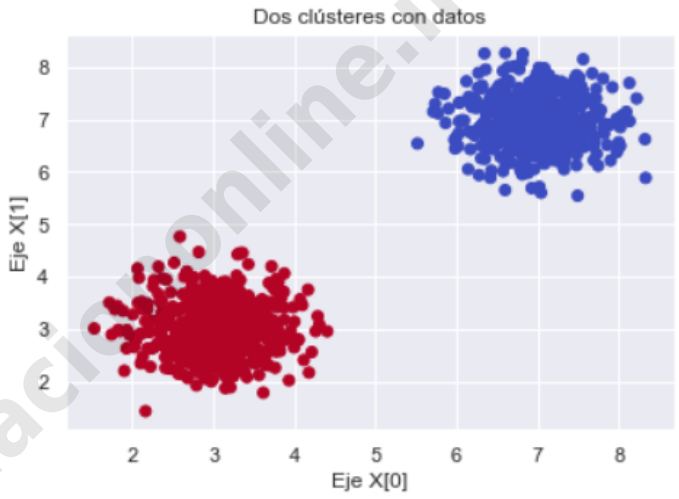
```
Estimación no. clusters: 2

Estimación no. noise points: 0
```

Por último, podemos generar un **diagrama de dispersión** para nuestros datos de entrenamiento. Dado que tenemos dos clústeres, utilizamos una función lambda simple que selecciona un color u otro. Si tenemos varios clústeres, podemos generalizar fácilmente esta función lambda con un

enfoque de diccionario.

```
# Dibujar scatter plot para datos de entrenamiento

colors = list(map(lambda x: '#3b4cc0' if x == 1 else '#b40426', labels))

plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)

plt.title('Dos clústeres con datos')

plt.xlabel('Eje X[0]')

plt.ylabel('Eje X[1]')

plt.show()
```

El **resultado final** son dos grupos, como se pretende:



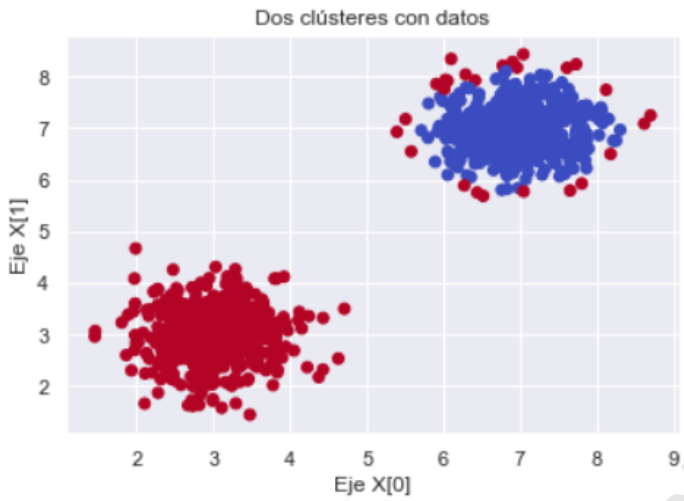
Si adaptamos el valor para ε y lo establecemos en 0.3, obtenemos diferentes resultados:

Estimación no. clusters: 3

Estimación no. noise points: 51

En concreto, el algoritmo es ahora capaz de detectar muestras ruidosas, como podemos ver en la siguiente imagen. Sin embargo, eliminar las muestras ruidosas después de realizar DBSCAN es fácil y requiere solo cuatro líneas de código adicional. Esto se debe a que DBSCAN establece las

etiquetas para muestras con ruido en ; esta es su forma de "señalar una etiqueta como ruidosa".-1

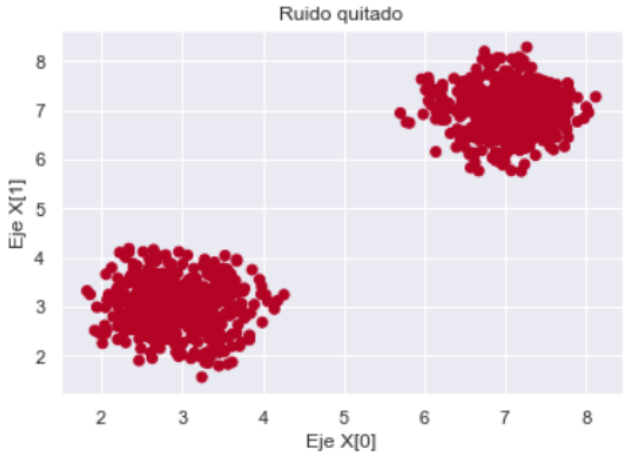


Al agregar las líneas antes de generar el diagrama de dispersión, se muestra que las muestras etiquetadas como ruido se eliminan del conjunto de datos.

Por esta razón, también podemos usar DBSCAN como un algoritmo de eliminación de ruido, por ejemplo, antes de aplicar la clasificación basada en SVM, para encontrar mejores límites de decisión.

```
# Quitar ruido

range_max = len(X)

X = np.array([X[i] for i in range(0, range_max) if labels[i] != -1])

labels = np.array([labels[i] for i in range(0, range_max) if labels[i] != -1])

# Dibujar scatter plot para datos de entrenamiento

colors = list(map(lambda x: '#000000' if x == -1 else '#b40426', labels))

plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)

plt.title(f'Ruido quitado')

plt.xlabel('Eje X[0]')

plt.ylabel('Eje X[1]')
```

plt.show()



A continuación, se **lista** todo el **código**:

```
# IMPORTAR PAQUETES

from sklearn.datasets import make_blobs

from sklearn.cluster import DBSCAN

import numpy as np

import matplotlib.pyplot as plt

# Configuración

num_samples_total = 1000

cluster_centers = [(3,3), (7,7)]

num_classes = len(cluster_centers)

epsilon = .3

min_samples = 13

# Generar datos

X, y = make_blobs(n_samples = num_samples_total, centers = cluster_centers, n_features =
```

```
num_classes, center_box=(0, 1), cluster_std = 0.5)

# Grabar datos si se considera necesario

np.save('./clusters.npy', X)

X = np.load('./clusters.npy')

# Computar DBSCAN

db = DBSCAN(eps=epsilon, min_samples=min_samples).fit(X)

labels = db.labels_

no_clusters = len(np.unique(labels) )

no_noise = np.sum(np.array(labels) == -1, axis=0)

print('Estimación no. clusters: %d' % no_clusters)

print('Estimación no. noise points: %d' % no_noise)

# Quitar ruido

range_max = len(X)

X = np.array([X[i] for i in range(0, range_max) if labels[i] != -1])

labels = np.array([labels[i] for i in range(0, range_max) if labels[i] != -1])

# Dibujar scatter plot para datos de entrenamiento

colors = list(map(lambda x: '#000000' if x == -1 else '#b40426', labels))

plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)

plt.title(f'Ruido quitado')

plt.xlabel('Eje X[0]')

plt.ylabel('Eje X[1]')
```

plt.show()

Algoritmo del modelo de mezcla gaussiana

Uno de los problemas con k-means es que los datos deben seguir un formato circular. La forma en que k-means calcula la distancia entre puntos de datos tiene que ver con una ruta circular, por lo que los datos no circulares no se agrupan correctamente.

Este es un problema que solucionan los **modelos de mezcla gaussianos**. No necesita datos de forma circular para que funcione bien.

El modelo de mezcla gaussiana utiliza múltiples distribuciones gaussianas para ajustar datos de forma arbitraria.

Hay varios modelos gaussianos únicos que actúan como capas ocultas en este modelo híbrido. El modelo calcula la probabilidad de que un punto de datos pertenezca a una distribución gaussiana específica y ese es el grupo en el que se ubicará.

Implementación:

Vamos a cargar las bibliotecas que necesitamos. Además de Pandas, Seaborn y numpy, utilizamos un par de módulos de scikit-learn.

```
import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs

from sklearn.mixture import GaussianMixture

import numpy as np

sns.set_context("talk", font_scale=1.5)
```

Usaremos la función de **make_blobs de sklearn.datasets** para crear un conjunto de datos simulado con 4 clústeres diferentes. El argumento centers=4 especifica cuatro clústeres. También

especificamos cómo de apretado debe ser el clúster usando cluster_std.

```
X, y = make_blobs(n_samples=500,

centers=4,

cluster_std=2,

random_state=2021)
```

make_blob funciones nos da los datos simulados como una matriz numpy y las etiquetas como vector. Almacenemos los datos como dataframe de Pandas.

```
data = pd.DataFrame(X)

data.columns=["X1","X2"]

data["cluster"]=y

data.head()
```

Los datos simulados tendrán este aspecto.

```
X1 X2 cluster

0 -0.685085 4.217225 0

1 11.455507 -5.728207 2

2 2.230017 5.938229 0

3 3.705751 1.875764 0

4 -3.478871 -2.518452 1
```

Con la **función GaussianMix() de scikit-learn**, podemos ajustar nuestros datos a los modelos de mezcla. Uno de los parámetros clave a utilizar al ajustar el modelo de mezcla gaussiana es el número de clústeres en el conjunto de datos.

Para este ejemplo, construyamos el modelo de mezcla gaussiana con 3 clústeres. Dado que

simulamos los datos con cuatro clústeres, sabemos que es incorrecto, pero sigamos adelante y ajustemos los datos con el modelo de mezcla gaussiana.

```
gmm = GaussianMixture(3,

covariance_type='full',

random_state=0).fit(data[["X1","X2"]])
```

Para los clústeres identificados, podemos obtener la ubicación de los medios utilizando el método "means_" en GaussianMixture.

```
gmm.means_

array([[-2.16398445, 4.84860401],

[ 9.97980069, -7.42299498],

[-7.28420067, -3.86530606]])
```

Usando la función predict(), también podemos predecir las etiquetas para los puntos de datos. En este ejemplo, obtenemos los valores predichos para los datos de entrada.

```
labels = gmm.predict(data[["X1","X2"]])
```

Agreguemos las etiquetas predichas a nuestro marco de datos.

```
data["predicted_cluster"]=labels
```

Finalmente visualizamos los datos coloreando los puntos de datos con etiquetas predichas.

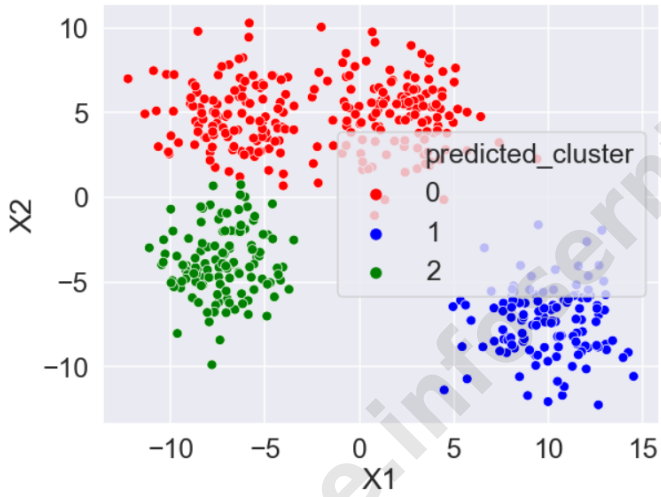
```
plt.figure(figsize=(9,7))

sns.scatterplot(data=data,

x="X1",

y="X2",

hue="predicted_cluster",
```

palette=["red","blue","green"])

plt.savefig("fitting_Gaussian_Mixture_Models_with_3_components_scikit_learn_Python.png",

format='png',dpi=150)

Podemos ver claramente que al ajustar el modelo con tres clústeres es incorrecto. El **modelo** ha agrupado dos **clústeres** en uno.



A menudo, el mayor desafío es que no sabremos los grupos de números en el conjunto de datos. Necesitamos identificar correctamente el número de grupos. Una de las formas en que podemos hacerlo es ajustar el modelo de mezcla gaussiana con un número múltiple de grupos, digamos que van de 1 a 20.

Y luego hacemos una **comparación de modelos** para encontrar qué modelo se ajusta primero a los datos. Por ejemplo, es un modelo de mezcla gaussiana con 4 clústeres que encajan mejor o un modelo con 3 clústeres que encajan mejor. Después podemos seleccionar el mejor modelo con cierto número de clústeres que se ajuste a los datos.

Las puntuaciones AIC o BIC se usan habitualmente para comparar modelos y seleccionar el mejor modelo que se ajuste a los datos. Una sola de las puntuaciones es lo suficientemente buena como para hacer una comparación de modelos. Sin embargo vamos a calcular ambas puntuaciones, solo para ver sus comportamientos.

Ajustamos los **datos** con el **modelo de mezcla** gaussiana con diferentes números de grupos.

```
n_components = np.arange(1, 21)

models = [GaussianMixture(n,

covariance_type='full', random_state=0).fit(X) for n in n_components]

models[0:5]

[GaussianMixture(random_state=0),

GaussianMixture(n_components=2, random_state=0),

GaussianMixture(n_components=3, random_state=0),

GaussianMixture(n_components=4, random_state=0),

GaussianMixture(n_components=5, random_state=0)]
```

Podemos calcular fácilmente los puntajes AIC / BIC con scikit-learn. Aquí usamos para uno de los modelos y calculamos las puntuaciones BIC y AIC.

```
models[0].bic(X)

6523.618150329507

models[0].aic(X)

6502.545109837397
```

Para comparar cómo cambia la puntuación **BIC/AIC** con respecto al número de componentes utilizados para construir el modelo de mezcla gaussiana, vamos a crear un marco de datos que contenga las puntuaciones BIC y AIC y el número de componentes.

```
gmm_model_comparisons=pd.DataFrame({"n_components" : n_components,

"BIC" : [m.bic(X) for m in models],

"AIC" : [m.aic(X) for m in models]})

gmm_model_comparisons.head()
```

Da como resultado la tabla:

n_components BIC AIC

0 1 6523.618150 6502.545110

1 2 6042.308396 5995.947707

2 3 5759.725951 5688.077613

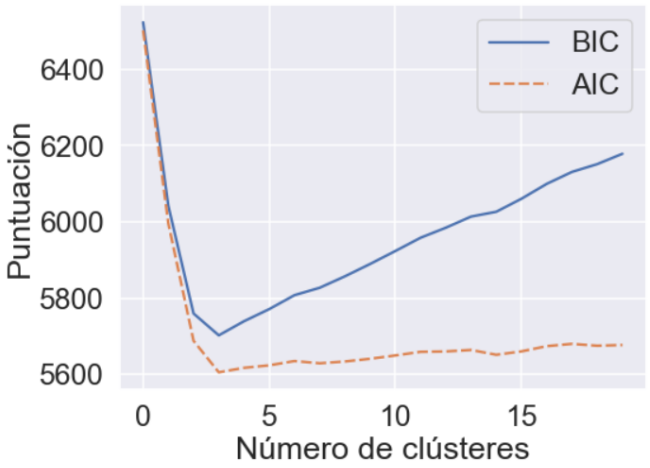
3 4 5702.439121 5605.503135

4 5 5739.478377 5617.254742

Ahora podemos hacer una **gráfica de línea de AIC/BIC** frente a los componentes numéricos.

```
plt.figure(figsize=(8,6))
sns.lineplot(data=gmm_model_comparisons[["BIC","AIC"]])
plt.xlabel("Número de clústeres")
plt.ylabel("Puntuación")
plt.savefig("GMM_model_comparison_with_AIC_BIC_Scores_Python.png",
format='png',dpi=150)
```

Podemos ver que las puntuaciones BIC y AIC están en el nivel más bajo cuando el número de componentes es 4. Por lo tanto, el modelo con n = 4 es el mejor modelo.



El **código completo** hasta ahora es el siguiente:

```
import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs

from sklearn.mixture import GaussianMixture

import numpy as np

sns.set_context("talk", font_scale=1.5)

X, y = make_blobs(n_samples=500,

centers=4,

cluster_std=2,

random_state=2021)

data = pd.DataFrame(X)

data.columns=["X1","X2"]

data["cluster"]=y
```

```
data.head()

gmm = GaussianMixture(3,

covariance_type='full',

random_state=0).fit(data[["X1","X2"]])

gmm.means_

labels = gmm.predict(data[["X1","X2"]])

data["predicted_cluster"]=labels

plt.figure(figsize=(9,7))

sns.scatterplot(data=data,

x="X1",

y="X2",

hue="predicted_cluster",

palette=["red","blue","green"])

plt.savefig("fitting_Gaussian_Mixture_Models_with_3_components_scikit_learn_Python.png",

format='png',dpi=150)

#Comparación de modelos

n_components = np.arange(1, 21)

models = [GaussianMixture(n,

covariance_type='full', random_state=0).fit(X) for n in n_components]

models[0:5]

[GaussianMixture(random_state=0),
```

```
GaussianMixture(n_components=2, random_state=0),

GaussianMixture(n_components=3, random_state=0),

GaussianMixture(n_components=4, random_state=0),

GaussianMixture(n_components=5, random_state=0)]

models[0].bic(X)

models[0].aic(X)

gmm_model_comparisons=pd.DataFrame({"n_components" : n_components,

"BIC" : [m.bic(X) for m in models],

"AIC" : [m.aic(X) for m in models]})

gmm_model_comparisons.head()

plt.figure(figsize=(8,6))

sns.lineplot(data=gmm_model_comparisons[["BIC","AIC"]])

plt.xlabel("Número de clústeres")

plt.ylabel("Puntuación")

plt.savefig("GMM_model_comparison_with_AIC_BIC_Scores_Python.png",

format='png',dpi=150)
```

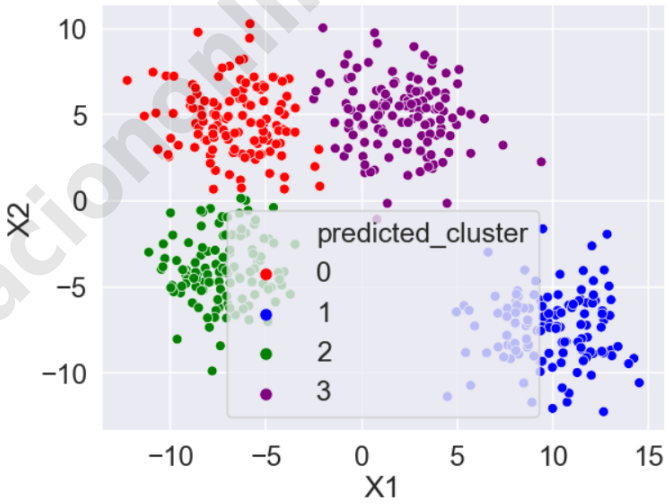
Ahora que sabemos el número de componentes necesarios para ajustarse al modelo, construyamos el modelo y extraigamos las etiquetas predichas para visualizar.

```
n=4

gmm = GaussianMixture(n, covariance_type='full', random_state=0).fit(data[["X1","X2"]])

labels = gmm.predict(data[["X1","X2"]])
```

```
data[["predicted_cluster"]]=labels
```

El scaterplot hecho con Seaborn resaltando los puntos de datos con las etiquetas predichas encaja perfectamente.

```
plt.figure(figsize=(9,7))

sns.scatterplot(data=data,

x="X1",

y="X2",

hue="predicted_cluster",

palette=["red","blue","green", "purple"])

plt.savefig("fitting_Gaussian_Mixture_Models_with_4_components_scikit_learn_Python.png",

format='png',dpi=150)
```



El **código completo** es el siguiente:

```
import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt
```

```
from sklearn.datasets import make_blobs

from sklearn.mixture import GaussianMixture

import numpy as np

sns.set_context("talk", font_scale=1.5)

X, y = make_blobs(n_samples=500,

centers=4,

cluster_std=2,

random_state=2021)

data = pd.DataFrame(X)

data.columns=["X1","X2"]

data["cluster"]=y

data.head()

n=4

gmm = GaussianMixture(n, covariance_type='full', random_state=0).fit(data[["X1","X2"]])

labels = gmm.predict(data[["X1","X2"]])

data["predicted_cluster"]=labels

plt.figure(figsize=(9,7))

sns.scatterplot(data=data,

x="X1",

y="X2",

hue="predicted_cluster",
```

```
palette=["red","blue","green", "purple"])

plt.savefig("fitting_Gaussian_Mixture_Models_with_4_components_scikit_learn_Python.png",

format='png',dpi=150)
```

Algoritmo BIRCH

El **algoritmo Balance Iterative Reducing and Clustering using Hierarchies (BIRCH)** funciona mejor en conjuntos de datos grandes que el algoritmo k-means.

Divide los datos en pequeños resúmenes que se agrupan en lugar de los puntos de datos originales. Los resúmenes contienen tanta información de distribución sobre los puntos de datos como sea posible.

Este algoritmo se usa habitualmente con otros algoritmos de agrupamiento porque las otras técnicas de agrupamiento se pueden usar en los resúmenes generados por BIRCH.

La principal desventaja del algoritmo BIRCH es que solo funciona con valores de datos numéricos. No puede usar esto para valores categóricos a menos que realice algunas transformaciones de datos.

Implementación:

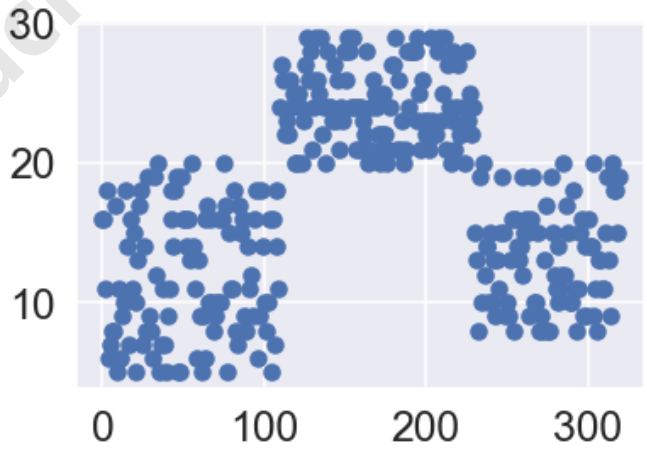
Al entrenar el modelo utilizando el algoritmo BIRCH, se crea una estructura de árbol con suficientes datos para asignar rápidamente cada punto de datos a un clúster. Al almacenar todos los puntos de datos en el árbol, este algoritmo permite el uso de memoria limitada mientras se trabaja en un conjunto de datos muy grande.

El **algoritmo BIRCH** comienza con un **valor de umbral**, luego aprende de los datos y luego inserta puntos de datos en el árbol. En el proceso, si se sale de la memoria mientras aprende los datos, aumenta el valor umbral y repite el proceso.

Como siempre, comenzamos importando las bibliotecas de **Python** necesarias y el conjunto de datos:

```
from sklearn.cluster import Birch
```

```
import numpy as np

import matplotlib.pyplot as plt

# Preparamos datos, creando datos aleatorios de agrupación en clústeres simples.

np.random.seed(12)

p1 = np.random.randint(5,21,110)

p2 = np.random.randint(20,30,120)

p3 = np.random.randint(8,21,90)

data = np.array(np.concatenate([p1, p2, p3]))

x_range = range(len(data))

x = np.array(list(zip(x_range, data))).reshape(len(x_range), 2)

# Mostramos los datos

plt.scatter(x[:,0], x[:,1])

plt.show()
```



A continuación, definiremos el método Birch y lo ajustaremos con x datos. Establecemos parámetros de branching_factor y parámetros de umbral (threshold). El branching_factor define el número de

subgrupos y el umbral establece el límite entre la muestra y el subgrupo.

bclust=Birch(branching_factor=100, threshold=.5).fit(x)

print(bclust)

Birch(branching_factor=100, compute_labels=True, copy=True, n_clusters=3,

threshold=0.5)

El método identifica el número de clústeres que se van a asignar. También se puede configurar manualmente. Ahora, podemos predecir x datos para obtener el id de clústeres de destino.

labels = bclust.predict(x)

Finalmente, verificaremos los puntos agrupados en una gráfica separándolos con diferentes colores.

A continuación, vemos el **código completo**:

from sklearn.cluster import Birch

import numpy as np

import matplotlib.pyplot as plt

Preparamos datos, creando datos aleatorios de agrupación en clústeres simples.

np.random.seed(12)

p1 = np.random.randint(5,21,110)

```
p2 = np.random.randint(20,30,120)

p3 = np.random.randint(8,21,90)

data = np.array(np.concatenate([p1, p2, p3]))

x_range = range(len(data))

x = np.array(list(zip(x_range, data))).reshape(len(x_range), 2)

# Mostramos los datos

plt.scatter(x[:,0], x[:,1])

plt.show()

# Clustering con Birch

bclust=Birch(branching_factor=100, threshold=.5).fit(x)

print(bclust)

Birch(branching_factor=100, compute_labels=True, copy=True, n_clusters=3,

threshold=0.5)

# Dibujar resultado

labels = bclust.predict(x)

plt.scatter(x[:,0], x[:,1], c=labels)

plt.show()
```

Algoritmo de agrupación en clústeres de propagación de afinidad (Affinity Propagation)

Este algoritmo de agrupación en clústeres es completamente diferente de los demás en la forma en que agrupa los datos.

Cada punto de datos se comunica con todos los demás puntos de datos para que los demás sepan

cómo de similares son y eso comienza a revelar los grupos en los datos. No hay que decirle a este algoritmo cuántos clústeres esperar en los parámetros de inicialización.

A medida que se envían mensajes entre **puntos de datos**, se encuentran conjuntos de **datos** llamados ejemplos y representan los grupos.

Un ejemplar se encuentra después de que los puntos de datos se hayan transmitido mensajes entre sí y formen un consenso sobre qué punto de datos representa mejor a un grupo.

Cuando no estemos seguros de cuántos clústeres esperar, como ocurre en un problema de visión por computadora, este es un gran algoritmo para comenzar.

Implementación:

Realizamos las importaciones habituales.

```
import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import make_blobs

from sklearn.cluster import AffinityPropagation
```

A continuación, añadimos algunas opciones de configuración: el número de muestras en total que generamos, los centros de los clústeres, así como el número de clases para las que generaremos muestras. Todos ellos se utilizarán en make_blobs, que genera los clústeres y los asigna X y targets, respectivamente.

Los guardamos con **Numpy** y posteriormente los cargamos y los asignamos a **X** de nuevo. Esas dos líneas de código no son necesarias para que el modelo se ejecute, pero si deseamos comparar entre configuraciones, es probable que no deseemos generar muestras al azar cada vez. Al guardarlos una vez, y posteriormente comentar las instrucciones sabe y **make_blobs** se cargarán desde el archivo.

```
# Opciones de configuración

num_samples_total = 50
```

```
cluster_centers = [(20,20), (4,4)]

num_classes = len(cluster_centers)

# Generar datos

X, targets = make_blobs(n_samples = num_samples_total, centers = cluster_centers, n_features =
num_classes, center_box=(0, 1), cluster_std = 1)

np.save('./clusters.npy', X)

X = np.load('./clusters.npy')
```

Después ajustamos los **datos** al **algoritmo** de propagación de afinidad, después de cargarlo, que solo necesita dos líneas de código. En otras dos líneas, derivamos características como los ejemplares y por consecuencia el número de clusters:

```
# Fit AFfinity Propagation with Scikit

afprop = AffinityPropagation(max_iter=250)

afprop.fit(X)

cluster_centers_indices = afprop.cluster_centers_indices_

n_clusters_ = len(cluster_centers_indices)
```

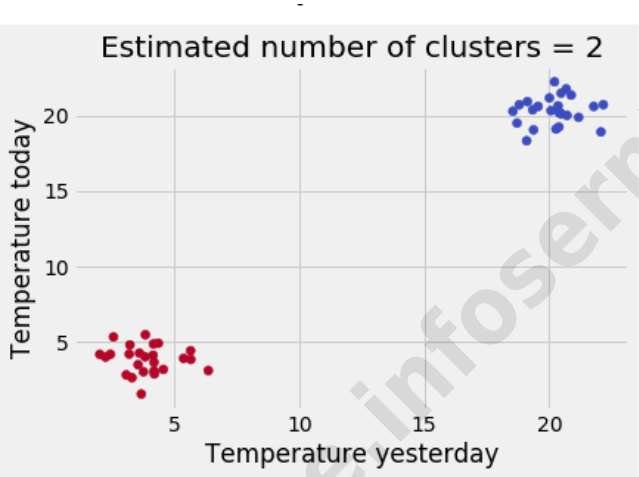
Finalmente, mediante el uso del algoritmo que encajamos, predecimos para todas nuestras muestras a qué clúster pertenecen:

```
# Predecir el clúster para todas las muestras

P = afprop.predict(X)
```

Y finalmente visualizar el resultado:

```
# Generar scatter plot para training data

colors = list(map(lambda x: '#3b4cc0' if x == 1 else '#b40426', P))
```

```
plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)

plt.title(f'Número estimado de clústers = {n_clusters_}')

plt.xlabel('Temperarura ayer')

plt.ylabel('Temperatura hoy')

plt.show()
```



A continuación, se lista el código completo:

```
import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import make_blobs

from sklearn.cluster import AffinityPropagation

# Opciones de configuración

num_samples_total = 50

cluster_centers = [(20,20), (4,4)]

num_classes = len(cluster_centers)

# Generar datos
```

```
X, targets = make_blobs(n_samples = num_samples_total, centers = cluster_centers, n_features =
num_classes, center_box=(0, 1), cluster_std = 1)

np.save('./clusters.npy', X)

X = np.load('./clusters.npy')

# Fit AFfinity Propagation with Scikit

afprop = AffinityPropagation(max_iter=250)

afprop.fit(X)

cluster_centers_indices = afprop.cluster_centers_indices_

n_clusters_ = len(cluster_centers_indices)

# Predecir el clúster para todas las muestras

P = afprop.predict(X)

# Generar scatter plot para training data

colors = list(map(lambda x: '#3b4cc0' if x == 1 else '#b40426', P))

plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)

plt.title(f'Número estimado de clústers = {n_clusters_}')

plt.xlabel('Temperarura ayer')

plt.ylabel('Temperatura hoy')

plt.show()
```

Algoritmo de agrupación en clústeres de cambio medio (Mean-Shift clustering)

Este es otro algoritmo que es particularmente útil para manejar imágenes y procesamiento de visión por computadora.

El desplazamiento medio es similar al **algoritmo BIRCH** porque también encuentra clústeres sin que se establezca un número inicial de clústeres.

Este es un algoritmo de agrupamiento jerárquico, pero la desventaja es que no escala bien cuando se trabaja con grandes conjuntos de datos.

Funciona iterando sobre todos los puntos de datos y los desplaza hacia el modo. El modo en este contexto es el área de alta densidad de puntos de datos en una región.

Es por eso que también se hace referencia a este algoritmo como el algoritmo de búsqueda de moda. Pasará por este proceso iterativo con cada punto de datos y los moverá más cerca de donde están otros puntos de datos hasta que todos los puntos de datos hayan sido asignados a un clúster.

Implementación:

Lo primero que hacemos es añadir las **importaciones para el código**.

```
import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import make_blobs

from sklearn.cluster import MeanShift, estimate_bandwidth
```

Usaremos Matplotlib para generar visualizaciones, Numpy para algún procesamiento de números y la funcionalidad Scikit-learn para generar el conjunto de datos (es decir, los blobs de datos no agrupados) y la operación de agrupación en clústeres real.

Una vez definidas las importaciones, podemos establecer las opciones de configuración:

```
# Configuración

num_samples_total = 10000

cluster_centers = [(5,5), (3,3), (1,1)]

num_classes = len(cluster_centers)
```

Generaremos 10000 muestras en total, en 3 clústeres.

Generamos los datos:

```
# Generar datos

X, targets = make_blobs(n_samples = num_samples_total, centers = cluster_centers, n_features = num_classes, center_box=(0, 1), cluster_std = 0.30)
```

Podemos hacer que **Scikit-learn gener los blobs** que queramos. Establecemos la configuración que acabamos de definir y establecemos una desviación estándar del clúster de 0,30. Esto puede ser casi cualquier cosa, es posible jugar con este valor para ver distintos resultados.

Es posible que desee guardar el conjunto de datos generados, y trabajar siempre con los mismos datos, con las siguientes líneas:

```
np.save('./clusters.npy', X)

X = np.load('./clusters.npy')
```

A continuación, llegaremos a la funcionalidad específica de Mean Shift. Primero, definimos lo que se conoce como el "ancho de banda" del algoritmo, como se puede ver aquí:

```
# Bandwith estimado

bandwidth = estimate_bandwidth(X, quantile=0.2, n_samples=500)
```

Mean Shift "mira a su alrededor" y determina la dirección a la que debe moverse una muestra, es decir, donde probablemente esté el centroide del clúster. Sin embargo, sería demasiado costoso computacionalmente hacerlo para todas las muestras, porque entonces el algoritmo se atascaría.

Es por eso que el "ancho de banda" ayuda: simplemente define un área alrededor de las muestras donde debe mirar el cambio medio para determinar la ruta más probable dada la estimación de densidad. Pero, ¿Cuál debería ser este valor de ancho de banda? Ahí es donde entra en juego, y estima el ancho de banda más adecuado en función del dataset.estimate_bandwidth

Inmediatamente usamos el **ancho de banda** en la instanciación del **algoritmo Mean Shift**, después de lo cual ajustamos los datos y generamos algunos datos consecuentes, como el número de

etiquetas:

```
# Fit Mean Shift con Scikit

meanshift = MeanShift(bandwidth=bandwidth)

meanshift.fit(X)

labels = meanshift.labels_

labels_unique = np.unique(labels)

n_clusters_ = len(labels_unique)
```

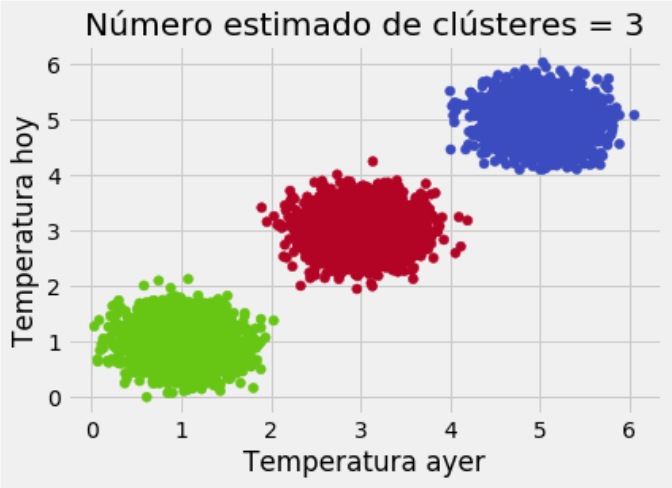
Luego, generamos predicciones para todas las muestras de nuestro conjunto de datos:

```
# Predecir el clúster para todas las muestras

P = meanshift.predict(X)
```

Y finalmente, generamos una **visualización** para ver si nuestra **operación de clustering** es exitosa:

```
# Generate scatter plot para training data

colors = list(map(lambda x: '#3b4cc0' if x == 1 else '#b40426' if x == 2 else '#67c614', P))

plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)

plt.title(f'Número estimado de clústeres = {n_clusters_}')

plt.xlabel('Temperatura ayer')

plt.ylabel('Temperatura hoy')

plt.show()
```

El código puede tardar unos minutos en ejecutarse



A continuación, se muestra el **código completo**:

```
import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import make_blobs

from sklearn.cluster import MeanShift, estimate_bandwidth

# Configuración

num_samples_total = 10000

cluster_centers = [(5,5), (3,3), (1,1)]

num_classes = len(cluster_centers)

# Generar datos

X, targets = make_blobs(n_samples = num_samples_total, centers = cluster_centers, n_features =
num_classes, center_box=(0, 1), cluster_std = 0.30)

np.save('./clusters.npy', X)

X = np.load('./clusters.npy')

# Bandwith estimado
```

```
bandwidth = estimate_bandwidth(X, quantile=0.2, n_samples=500)

# Fit Mean Shift con Scikit

meanshift = MeanShift(bandwidth=bandwidth)

meanshift.fit(X)

labels = meanshift.labels_

labels_unique = np.unique(labels)

n_clusters_ = len(labels_unique)

# Predecir el clúster para todas las muestras

P = meanshift.predict(X)

# Generate scatter plot para training data

colors = list(map(lambda x: '#3b4cc0' if x == 1 else '#b40426' if x == 2 else '#67c614', P))

plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)

plt.title(f'Número estimado de clústeres = {n_clusters_}')

plt.xlabel('Temperatura ayer')

plt.ylabel('Temperatura hoy')

plt.show()
```


Algoritmo OPTICS

OPTICS significa Ordenar puntos para identificar la **estructura de agrupamiento**. Es un algoritmo basado en densidad similar a DBSCAN, pero es mejor porque puede encontrar agrupaciones significativas en datos que varían en densidad. Lo hace ordenando los puntos de datos de modo que los puntos más cercanos sean vecinos en el ordenamiento.

Esto facilita la detección de diferentes grupos de densidad. El **algoritmo OPTICS** solo procesa cada

punto de datos una vez, similar a DBSCAN (aunque se ejecuta más lento que DBSCAN). También hay una distancia especial almacenada para cada punto de datos que indica que un punto pertenece a un grupo específico.

Implementación:

Con el siguiente código, podemos realizar **clustering** basado en OPTICS en un conjunto de datos aleatorio similar a un blob. Funciona de la siguiente manera:

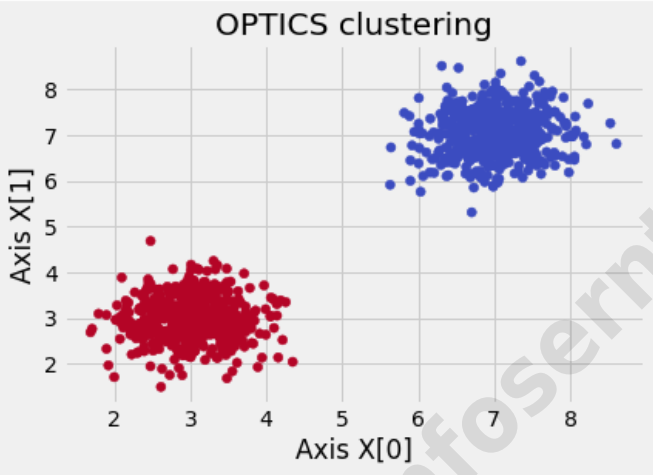
- En primer lugar, hacemos todas las importaciones; para generar los datos, para la agrupación en clústeres, y NumPy y Matplotlib para el procesamiento y la visualización de números, respectivamente.
- Especificamos una gama de opciones de configuración. Generaremos 1000 muestras en total alrededor de dos centros, de modo que obtendremos dos blobs de datos. Establecimos epsilon y min_samples a valores que derivamos durante las pruebas, así como el método para agrupar y la métrica de distancia.
- La distancia de Minkowski es la métrica predeterminada.
- A continuación generamos datos: dos blobs de datos, con .make_blobs
- En base a estos datos, realizamos clustering basado en OPTICS, con epsilon, número mínimo de muestras, método de cluster y métrica definida. Inmediatamente ajustamos los datos para que se generen los clústeres.
- Luego imprimimos información sobre el número de clústeres y muestras ruidosas, y finalmente generamos un diagrama de dispersión.

```
from sklearn.datasets import make_blobs

from sklearn.cluster import OPTICS

import numpy as np

import matplotlib.pyplot as plt

# Configuración

num_samples_total = 1000
```

```
cluster_centers = [(3,3), (7,7)]

num_classes = len(cluster_centers)

epsilon = 2.0

min_samples = 22

cluster_method = 'xi'

metric = 'minkowski'

# Generar datos

X, y = make_blobs(n_samples = num_samples_total, centers = cluster_centers, n_features =
num_classes, center_box=(0, 1), cluster_std = 0.5)

# Computar OPTICS

db = OPTICS(max_eps=epsilon, min_samples=min_samples, cluster_method=cluster_method,
metric=metric).fit(X)

labels = db.labels_

no_clusters = len(np.unique(labels) )

no_noise = np.sum(np.array(labels) == -1, axis=0)

print('Estimado no. clusters: %d' % no_clusters)

print('Estimado no. noise points: %d' % no_noise)

# Generar scatter plot para training data

colors = list(map(lambda x: '#3b4cc0' if x == 1 else '#b40426', labels))

plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)

plt.title(f'OPTICS clustering')
```

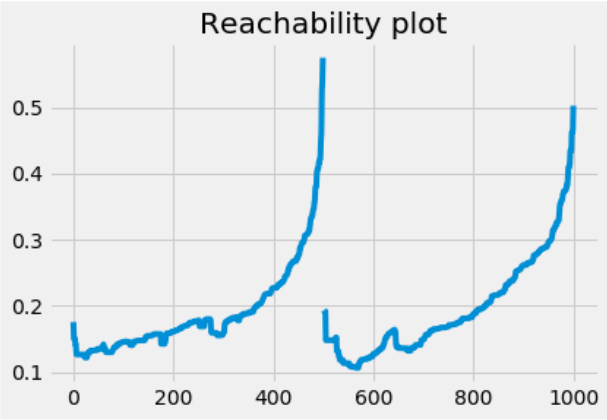
```
plt.xlabel('Axis X[0]')

plt.ylabel('Axis X[1]')

plt.show()
```

```
Estimado no. clusters: 3
Estimado no. noise points: 475
```



También podemos generar fácilmente la gráfica de accesibilidad:

```
# Generar gráfica de accesibilidad

reachability = db.reachability_[db.ordering_]

plt.plot(reachability)

plt.title('Reachability plot')

plt.show()
```



Algoritmo de agrupamiento de jerarquía aglomerativa (Agglomerative Hierarchy clustering)

Este es el tipo más común de algoritmo de **agrupamiento jerárquico**. Se utiliza para agrupar objetos en grupos en función de su similitud entre sí.

Esta es una forma de agrupamiento de abajo hacia arriba, donde cada punto de datos se asigna a su propio grupo. Luego, esos grupos se unen.

En cada iteración, los clústeres similares se fusionan hasta que todos los puntos de datos son parte de un gran clúster raíz.

La **agrupación aglomerativa** es mejor para encontrar agrupaciones pequeñas. El resultado final parece un **dendrograma** para que pueda visualizar fácilmente los grupos cuando finaliza el algoritmo.

Un dendrograma (o diagrama de árbol) es una estructura de red. Está constituido por un nodo raíz que da a luz a varios nodos conectados por bordes o ramas. Los últimos nodos de la jerarquía se llaman hojas.

Implementación:

Los pasos para realizar el algoritmo son los siguientes:

- Paso 1: tratar cada punto de datos como un solo clúster. Por lo tanto, tendremos, K grupos al principio. El número de puntos de datos también será K al principio.
- Paso 2: necesitamos formar un gran clúster uniendo dos puntos de datos de armario. Esto dará como resultado un total de clústeres K-1.
- Paso 3: para formar más clústeres necesitamos unir los dos puntos más cercanos. Esto dará como resultado un total de clústeres K-2.
- Paso 4: para formar un gran clúster, repetimos los tres pasos anteriores hasta que K se convierta en 0, es decir, no queden más puntos de datos para unirse.
- Paso 5: por último, después de hacer un solo clúster grande, se utilizarán dendrogramas para dividir en múltiples grupos dependiendo del problema.

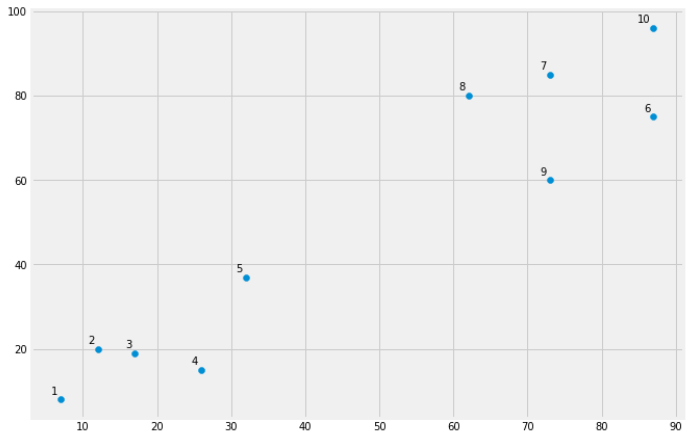
Como discutimos en el último paso, el papel del **dendrograma** comienza una vez que se forma el gran cluste. El dendrograma se utilizará para dividir los clústeres en múltiples grupos de puntos de datos relacionados dependiendo de nuestro problema.

Comenzamos con la importación de las **bibliotecas requeridas** de la siguiente manera:

```
%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np
```

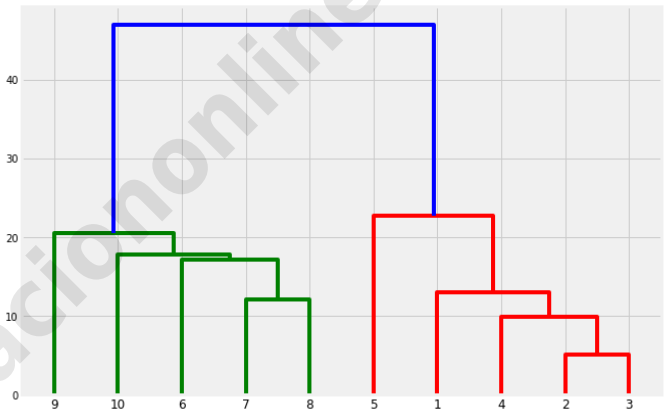
A continuación, trazaremos los puntos de datos que hemos tomado para este ejemplo−

```
X = np.array([[7,8],[12,20],[17,19],[26,15],[32,37],[87,75],[73,85], [62,80],[73,60],[87,96],])

labels = range(1, 11)

plt.figure(figsize=(10, 7))

plt.subplots_adjust(bottom=0.1)

plt.scatter(X[:,0],X[:,1], label='True Position')

for label, x, y in zip(labels, X[:, 0], X[:, 1]):

plt.annotate(label,xy=(x, y), xytext=(-3, 3),textcoords='offset points', ha='right', va='bottom')

plt.show()
```

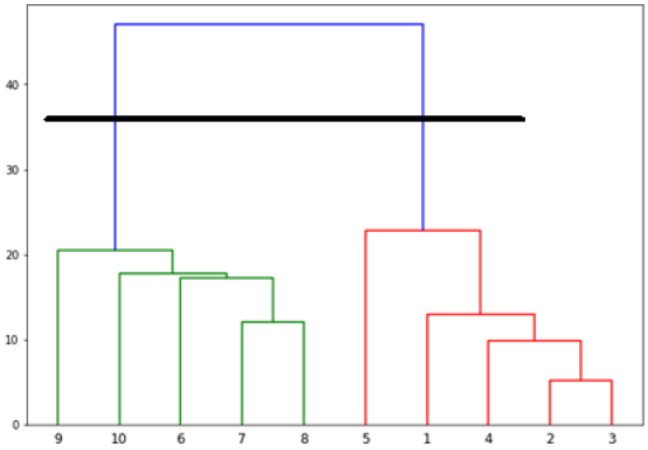


A partir del diagrama anterior, es muy fácil ver que tenemos dos clústeres en nuestros puntos de datos, pero en los datos del mundo real, puede haber miles de clústeres. A continuación, trazaremos los dendrogramas de nuestros puntos de datos utilizando la biblioteca Scipy.

```
from scipy.cluster.hierarchy import dendrogram, linkage

from matplotlib import pyplot as plt

linked = linkage(X, 'single')

labelList = range(1, 11)

plt.figure(figsize=(10, 7))

dendrogram(linked, orientation='top',labels=labelList,
distance_sort='descending',show_leaf_counts=True)

plt.show()
```



Ahora, una vez que se forma el gran cúmulo, se selecciona la **distancia vertical** más larga. A continuación, se dibuja una línea vertical a través de ella como se muestra en el siguiente diagrama. Como la línea horizontal cruza la línea azul en dos puntos, el número de grupos sería de dos.



A continuación, debemos importar la clase para la agrupación en clústeres y llamar a su método fit_predict para predecir el clúster. Estamos importando la clase AgglomerativeClustering de la biblioteca sklearn.cluster.

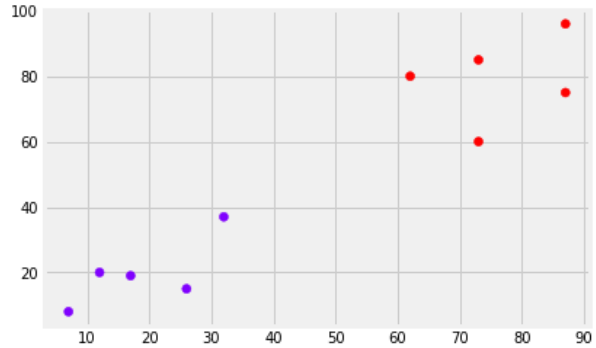
```
from sklearn.cluster import AgglomerativeClustering

cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean', linkage='ward')

cluster.fit_predict(X)
```

A continuación, trazamos el clúster

```
plt.scatter(X[:,0],X[:,1], c=cluster.labels_, cmap='rainbow')
```

El diagrama muestra los dos clústeres de nuestros puntos de datos.

```
Out[22]: <matplotlib.collections.PathCollection at 0x1789a855c08>
```



A continuación, se incluye el **código completo**:

```
%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

X = np.array([[7,8],[12,20],[17,19],[26,15],[32,37],[87,75],[73,85], [62,80],[73,60],[87,96],])

labels = range(1, 11)

plt.figure(figsize=(10, 7))

plt.subplots_adjust(bottom=0.1)

plt.scatter(X[:,0],X[:,1], label='True Position')

for label, x, y in zip(labels, X[:, 0], X[:, 1]):

plt.annotate(label,xy=(x, y), xytext=(-3, 3),textcoords='offset points', ha='right', va='bottom')

plt.show()

#Dibujar dendrograma

from scipy.cluster.hierarchy import dendrogram, linkage

from matplotlib import pyplot as plt

linked = linkage(X, 'single')

labelList = range(1, 11)

plt.figure(figsize=(10, 7))

dendrogram(linked, orientation='top',labels=labelList,

distance_sort='descending',show_leaf_counts=True)

plt.show()

#Predecir el clúster
```

```
from sklearn.cluster import AgglomerativeClustering

cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean', linkage='ward')

cluster.fit_predict(X)

plt.scatter(X[:,0],X[:,1], c=cluster.labels_, cmap='rainbow')
```

Con el aprendizaje semi-supervisado, tenemos un gran conjunto de datos donde algunos de los datos están etiquetados pero la mayoría no.

- Verdadero.
- Falso.

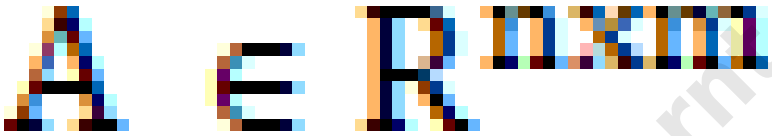
Las puntuaciones AIC o BIC se usan habitualmente para comparar modelos y seleccionar el mejor modelo que se ajuste a los datos. Una sola de las puntuaciones es lo suficientemente buena como para hacer una comparación de modelos. Sin embargo vamos a calcular ambas puntuaciones, solo para ver sus comportamientos.

- Verdadero.
- Falso.

Biclustering

Biclustering es un método de agrupamiento que opera simultáneamente en dos niveles que están correlacionados por la presencia de un medio (por ejemplo, clientes y productos por calificación). Biclustering tiene como objetivo encontrar las regiones donde el medio es cohesivo (por ejemplo, la calificación es alta o baja) reordenando la estructura de ambos niveles.

Biclustering opera en una matriz

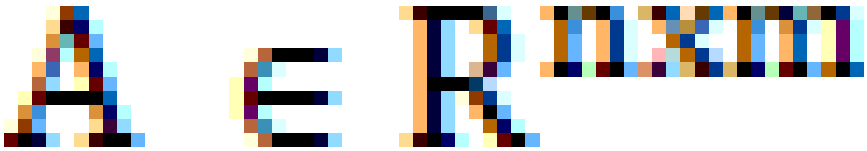


A indica una calificación (o cero) para el producto dado por el cliente CI. A tiene una estructura de tablero de ajedrez subyacente, donde las regiones compactas (biclusters) representan submatrices.

Biclustering espectral es un algoritmo desarrollado por Kluger et al. (2003) para clasificar genes y condiciones. El Biclustering espectral se basa en la descomposición de valores singulares (SVD) y se aplicó inicialmente a tareas de bioinformática, pero también ha encontrado aplicaciones en otros campos.

El **algoritmo** consta de **cinco pasos**:

- Bistocaización: una fase iterativa de preprocesamiento donde los valores de aij se ajustan para que todas las sumas de filas y columnas sumen un valor común constante (generalmente 1).
- Descomposición de la matriz bistocástica



Los valores singulares se ordenan en orden descendente y los vectores singulares se reorganizan en consecuencia.

- Ranking de los vectores singulares analizando su similitud con

la estructura del tablero de ajedrez se resaltará con biclustering.

- Proyección del conjunto de datos en el subespacio abarcado por las columnas de la matriz

- Aplicación de K-Means para encontrar el etiquetado delkracimos. Esta operación produce dos vectores de etiquetas,

A_b utilizando ***SVD***. b se descompone en vectores sing***ulares izquierdo*** y ***derecho*** (***A_b*** ***A_b***^T y ***A_b***^***T*** ***A_b***

Implementación de Biclustering espectral en Python.

Para implementar **biclustering espectral**, primero necesitamos un conjunto de **datos**. Por razones prácticas, podemos generar un conjunto de datos artificiales que consta de:

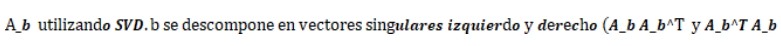
- 100 usuarios.
- 100 transacciones (compras).
- 100 productos.

- clasificaciones en el rango 1-10 (10 posibles biclusters diferentes).

```
import numpy as np

nb_users = 100

nb_products = 100

items = [i for i in range(nb_products)]

transactions = []

ratings = np.zeros(shape=(nb_users, nb_products), dtype=np.int)

for i in range(nb_users):

n_items = np.random.randint(2, 60)

transaction = tuple(np.random.choice(items, replace=False, size=n_items))

transactions.append(list(map(lambda x: "P{}".format(x+1), transaction)))

for t in transaction:

rating = np.random.randint(1, 11)

ratings[i, t] = rating
```

Podemos visualizar las calificaciones generadas en un **mapa de calor**:

```
import seaborn as sns

sns.heatmap(ratings, center=0)
```



Ahora podemos implementar el algoritmo de **biclustering espectral con scikit-learn**:

```
from sklearn.cluster import SpectralBiclustering
```

```
sbc = SpectralBiclustering(n_clusters = 10,

n_best = 5, #projectar el dataset en el top 5 de vectore singulares

svd_method = "arpack", #bueno para matrices pequeñas/medianas

random_state = 1000)

sbc.fit(ratings)
```

A continuación, necesitamos calcular la matriz final utilizando el producto externo de los índices de fila y columna ordenados:

```
rc = np.outer(np.sort(sbc.row_labels_) + 1,

np.sort(sbc.column_labels_) + 1)
```

También podemos visualizar la matriz reorganizada en un mapa de calor:

```
sns.heatmap(rc)
```

Finalmente, podemos usar la matriz para un caso de uso práctico de marketing: determinar el grupo de usuarios que calificaron un grupo de cinco productos para enviar a esos usuarios un boletín con recomendaciones de productos.

Para hacer esto, necesitamos seleccionar todas las **filas** y **columnas** asociadas con los **biclusters** con un índice de 5 (porque 0 significa sin calificación).

```
print("Usuarios: {}".format(np.where(sbc.rows_[8, :] == True)))

print("Productos: {}".format(np.where(sbc.columns_[8, :] == True)))
```

El código anterior genera el siguiente resultado:

```
Usuarios: (array([ 4, 16, 21, 31, 46, 50, 51, 54, 55, 65, 69, 73, 86, 93],

dtype=int64),)

Productos: (array([13, 29], dtype=int64),)
```

Esto significa que debemos verificar los productos en del array Productos, seleccionar otros productos que sean similares a esos y enviarlos a los usuarios del array Usuarios.

El código completo es el siguiente:

```
import numpy as np

nb_users = 100

nb_products = 100

items = [i for i in range(nb_products)]

transactions = []

ratings = np.zeros(shape=(nb_users, nb_products), dtype=np.int)

for i in range(nb_users):

n_items = np.random.randint(2, 60)

transaction = tuple(np.random.choice(items, replace=False, size=n_items))

transactions.append(list(map(lambda x: "P{}".format(x+1), transaction)))

for t in transaction:

rating = np.random.randint(1, 11)

ratings[i, t] = rating

#mapa de calor

import seaborn as sns

sns.heatmap(ratings, center=0)

#Biclustering

from sklearn.cluster import SpectralBiclustering
```

```
sbc = SpectralBiclustering(n_clusters = 10,

n_best = 5, #projectar el dataset en el top 5 de vectore singulares

svd_method = "arpack", #bueno para matrices pequeñas/medianas

random_state = 1000)

sbc.fit(ratings)

rc = np.outer(np.sort(sbc.row_labels_) + 1,

np.sort(sbc.column_labels_) + 1)

#mapa de calor reorganizado

sns.heatmap(rc)

# Seleccionar usuarios

print("Usuarios: {}".format(np.where(sbc.rows_[8, :] == True)))

print("Productos: {}".format(np.where(sbc.columns_[8, :] == True)))
```

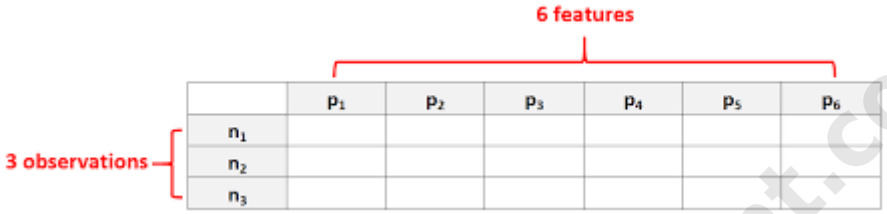

Relaciona los elementos de la columna Derecha con la columna Izquierda

Biclustering espectral	1
Biclustering opera en una matriz Biclustering	2

	es un algoritmo desarrollado por Kluger et al. (2003) para clasificar genes y condiciones.
	Indica una calificación (o cero) para el producto dado por el cliente CI. A tiene una estructura de tablero de ajedrez subyacente, donde las regiones compactas (biclusters) representan submatrices.

Manifolds. Reducción de la dimensionalidad

Los **conjuntos de datos de alta dimensión** pueden ser muy difíciles de visualizar. Si bien los datos en dos o tres dimensiones pueden trazarse para mostrar la estructura inherente de los datos, los gráficos de alta dimensión equivalentes son mucho menos intuitivos. Para ayudar a visualizar la estructura de un conjunto de datos, la dimensión debe reducirse de alguna manera.



La forma más sencilla de lograr esta reducción de dimensionalidad es tomando una proyección aleatoria de los datos. Aunque esto permite cierto grado de visualización de la estructura de datos, la aleatoriedad de la elección deja mucho que desear. En una proyección aleatoria, es probable que se pierda la estructura más interesante dentro de los datos.

Para abordar esta cuestión, se han diseñado varios marcos de reducción de dimensionalidad lineal supervisados y no supervisados, como el análisis de componentes principales (PCA), el análisis de componentes independientes, el análisis discriminante lineal y otros. Estos algoritmos definen **rúbricas** específicas para elegir una proyección lineal "interesante" de los datos. Estos métodos pueden ser poderosos, pero a menudo pierden una estructura no lineal importante en los datos.

El **aprendizaje Manifold (múltiple)** se puede considerar como un intento de generalizar marcos lineales como PCA para que sean sensibles a la estructura no lineal de los datos. Aunque existen variantes supervisadas, el típico problema de aprendizaje múltiple no está supervisado: aprende la estructura de alta dimensión de los datos a partir de los datos mismos, sin el uso de clasificaciones predeterminadas.

Tipos de aprendizaje Manifold

Se distinguen los siguientes tipos:

Isomap

Uno de los primeros enfoques para el aprendizaje múltiple es el algoritmo Isomap, abreviatura de Isometric Mapping. Isomap puede verse como una extensión del Escalado multidimensional (MDS) o Kernel PCA. Isomap busca una incrustación de menor dimensión que mantenga las distancias geodésicas entre todos los puntos.

Incrustación localmente lineal (LLE)

La incrustación localmente lineal (LLE) busca una proyección de menor dimensión de los datos que preserva las distancias dentro de los vecindarios locales. Se puede considerar como una serie de análisis de componentes principales locales que se comparan globalmente para encontrar la mejor integración no lineal.

Incrustación lineal localmente modificada

Un problema bien conocido con LLE es el problema de regularización. Cuando el número de vecinos es mayor que el número de dimensiones de entrada, la matriz que define cada vecindario local es deficiente en rango. Para abordar esto, LLE estándar aplica un parámetro de regularización arbitrario, que se elige en relación con la traza de la matriz de peso local. Aunque se puede demostrar formalmente que como r->0, la solución converge a la incrustación deseada, no hay garantía de que se encontrará la solución óptima para r>0. Este problema se manifiesta en incrustaciones que distorsionan la geometría subyacente del manifold.

Eigenmapping de Hesse (HLLE)

El mapeo propio de Hessian (también conocido como LLE basado en Hessian: HLLE) es otro método para resolver el problema de regularización de LLE. Gira en torno a una forma cuadrática basada en Hessian para cada vecino que se utiliza para recuperar la estructura lineal local.

Incrustación espectral

La incrustación espectral es un método para calcular una incrustación no lineal. Scikit-learn implementa laplacian Eigenmaps, que encuentra una representación de baja dimensión de los datos utilizando una descomposición espectral del gráfico laplaciano. El gráfico generado se puede considerar como una aproximación discreta del manifold de baja dimensión en el espacio de alta dimensión. La minimización de una función de coste basada en el gráfico asegura que los puntos

cercanos entre sí en el manifold se mapeen cerca entre sí en el espacio de baja dimensión, preservando las distancias locales.

Alineación del espacio tangente local (LTSA)

Aunque técnicamente no es una variante de LLE, la alineación del espacio tangente local (LTSA) es algorítmicamente lo suficientemente similar a LLE que puede incluirse en esta categoría. En lugar de centrarse en preservar las distancias del vecindario como en LLE, LTSA busca caracterizar la geometría local en cada vecindario a través de su espacio tangente y realiza una optimización global para alinear estos espacios tangentes locales para aprender la incrustación.

Escalado multidimensional (MDS)

El escalado multidimensional (MDS) busca una representación de baja dimensión de los datos en la que las distancias respeten bien las distancias en el espacio original de alta dimensión.

En general, MDS es una técnica que se utiliza para analizar datos de similitud o disimilitud. Intenta modelar datos de similitud o disimilitud como distancias en espacios geométricos. Los datos pueden ser calificaciones de similitud entre objetos, frecuencias de interacción de moléculas o índices comerciales entre países.

Existen dos tipos de algoritmo MDS: métrico y no métrico.

Incrustación de vecinos estocásticos distribuidos en t (t-SNE)

t-distributed Stochastic Neighbor Embedding - t-SNE (TSNE) convierte afinidades de puntos de datos en probabilidades. Las afinidades en el espacio original están representadas por probabilidades conjuntas de Gauss y las afinidades en el espacio incrustado están representadas por distribuciones t de Student. Esto permite que t-SNE sea particularmente sensible a la estructura

local y tiene algunas otras ventajas sobre las técnicas existentes:

- Revela la estructura a muchas escalas en un solo mapa.
- Revelardatos que se encuentran en múltiples, diferentes, múltiples o agrupaciones.
- Reduce la tendencia a agrupar puntos en el centro.

Si bien Isomap, LLE y las variantes son las más adecuadas para desplegar una única variedad continua de baja dimensión, t-SNE se centrará en la estructura local de los datos y tenderá a extraer grupos de muestras locales agrupados como se destaca en el ejemplo de la curva S. Esta capacidad de agrupar muestras en función de la estructura local podría ser beneficiosa para desenredar visualmente un conjunto de datos que comprenda varias variedades a la vez, como es el caso en el conjunto de datos de dígitos.

Barnes-Hut t-SNE

El t-SNE de Barnes-Hut suele ser mucho más lento que otros algoritmos de aprendizaje múltiples. La optimización es bastante difícil y el cálculo del gradiente es O[dNlog(N)], donde d es el número de dimensiones de salida y N es el número de muestras. El método de Barnes-Hut mejora el método exacto donde la complejidad de t-SNE es O[dN2], pero tiene otras diferencias notables:

- La implementación de Barnes-Hut solo funciona cuando la dimensionalidad objetivo es 3 o menos. El caso 2D es típico cuando se crean visualizaciones.
- Barnes-Hut solo funciona con datos de entrada densos. Las matrices de datos dispersos solo se pueden incrustar con el método exacto o se pueden aproximar mediante una proyección densa de rango bajo, por ejemplo, utilizandoTruncatedSVD.
- Barnes-Hut es una aproximación del método exacto. La aproximación se parametriza con el parámetro de ángulo, por lo tanto, el parámetro de ángulo no se utiliza cuando método = "exacto"
- Barnes-Hut es significativamente más escalable. Barnes-Hut se puede utilizar para incrustar cientos de miles de puntos de datos, mientras que el método exacto puede manejar miles de muestras antes de volverse intratable computacionalmente.

Para fines de visualización (que es el caso de uso principal de t-SNE), se recomienda encarecidamente utilizar el método de Barnes-Hut. El método t-SNE exacto es útil para verificar las propiedades teóricas de la incrustación posiblemente en un espacio dimensional superior, pero se limita a pequeños conjuntos de datos debido a restricciones computacionales.

Hay que tener en cuenta que las **etiquetas de dígitos** coinciden aproximadamente con la agrupación natural encontrada por t-SNE, mientras que la proyección 2D lineal del modelo PCA produce una representación donde las regiones de etiquetas se superponen en gran medida. Esta es una pista sólida de que estos datos pueden separarse bien mediante métodos no lineales que se centran en la estructura local (por ejemplo, una SVM con un kernel RBF gaussiano).

Sin embargo, no poder visualizar grupos bien separados etiquetados homogéneamente con t-SNE en 2D no implica necesariamente que los datos no puedan ser clasificados correctamente por un modelo supervisado. Puede darse el caso de que dos dimensiones no sean lo suficientemente altas para representar con precisión la estructura interna de los datos.

Implementación de Manifold

Vamos a ver una ilustración de la reducción de dimensionalidad en el conjunto de datos de la curva S con varios métodos de aprendizaje.

Hay que tener en cuenta que el propósito del MDS es encontrar una representación de baja dimensión de los datos (en este caso 2D) en la que las distancias respeten bien las distancias en el espacio original de alta dimensión, a diferencia de otros algoritmos de aprendizaje múltiple, no busca una representación isotrópica de los datos en el espacio de baja dimensión.

El **código** que muestra todos los **métodos** aplicados a la **curva S** tridimensional es el siguiente:

```
from collections import OrderedDict

from functools import partial

from time import time

import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D

from matplotlib.ticker import NullFormatter

from sklearn import manifold, datasets

Axes3D
```

```
n_points = 1000

X, color = datasets.make_s_curve(n_points, random_state=0)

n_neighbors = 10

n_components = 2

# Create figure

fig = plt.figure(figsize=(15, 8))

fig.suptitle(

"Manifold Learning con %i puntos, %i vecinos" % (1000, n_neighbors), fontsize=14

)

# 3d scatter plot

ax = fig.add_subplot(251, projection="3d")

ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)

ax.view_init(4, -72)

# Configurar métodos manifold

LLE = partial(

manifold.LocallyLinearEmbedding,

n_neighbors=n_neighbors,

n_components=n_components,

eigen_solver="auto",

)

methods = OrderedDict()
```

```
methods["LLE"] = LLE(method="standard")

methods["LTSA"] = LLE(method="ltsa")

methods["Hessian LLE"] = LLE(method="hessian")

methods["Modified LLE"] = LLE(method="modified")

methods["Isomap"] = manifold.Isomap(n_neighbors=n_neighbors, n_components=n_components)

methods["MDS"] = manifold.MDS(n_components, max_iter=100, n_init=1)

methods["SE"] = manifold.SpectralEmbedding(

n_components=n_components, n_neighbors=n_neighbors

)

methods["t-SNE"] = manifold.TSNE(n_components=n_components, init="pca", random_state=0)

# Dibujar resultados

for i, (label, method) in enumerate(methods.items()):

t0 = time()

Y = method.fit_transform(X)

t1 = time()

print("%s: %.2g sec" % (label, t1 - t0))

ax = fig.add_subplot(2, 5, 2 + i + (i > 3))

ax.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)

ax.set_title("%s (%.2g sec)" % (label, t1 - t0))

ax.xaxis.set_major_formatter(NullFormatter())

ax.yaxis.set_major_formatter(NullFormatter())
```

```
ax.axis("tight")

plt.show()
```

Los conjuntos de datos de alta dimensión pueden ser muy difíciles de visualizar. Si bien los datos en dos o tres dimensiones pueden trazarse para mostrar la estructura inherente de los datos, los gráficos de alta dimensión equivalentes son mucho menos intuitivos. Para ayudar a visualizar la estructura de un conjunto de datos, la dimensión debe reducirse de alguna manera.

- Verdadero.
- Falso.

Análisis de la cesta.

Hay muchas herramientas de análisis de datos disponibles para el analista de Python y puede ser difícil saber cuáles usar en una situación particular. Una técnica útil (pero algo pasada por alto) se llama **análisis de asociación** que intenta encontrar patrones comunes de elementos en grandes conjuntos de datos.

Una aplicación específica a menudo se llama **análisis de cesta** de mercado. El ejemplo más conocido de análisis de cesta de mercado es el llamado caso de "cerveza y pañales". La historia básica es que un gran minorista pudo extraer sus datos de transacciones y encontrar un patrón de compra inesperado de personas que compraban cerveza y pañales para bebés al mismo tiempo.

Probablemente esta historia sea una leyenda urbana de datos. Sin embargo, es un ejemplo ilustrativo del tipo de información que se pueden obtener al extraer datos transaccionales.

Si bien este tipo de asociaciones se utilizan normalmente para examinar las transacciones de venta, el análisis básico se puede aplicar a otras situaciones como el seguimiento del flujo de clics, el pedido de piezas de repuesto y los motores de recomendación en línea, solo por nombrar algunos.

Como hemos visto en tantos casos, lo primero sería mirar scikit-learn para un algoritmo ya hecho. Sin embargo, scikit-learn no es compatible con este algoritmo. Afortunadamente, la biblioteca MLxtend de Sebastian Raschka tiene una implementación del algoritmo llamada **apriori** para extraer conjuntos de elementos frecuentes para su posterior análisis.

Análisis de asociación

En el mundo actual, existen muchas formas complejas de analizar datos (clustering, regresión, Redes Neuronales, Bosques Aleatorios, SVM,etc.). El desafío con muchos de estos enfoques es que pueden ser difíciles de ajustar, difíciles de interpretar y requieren bastante preparación de datos e ingeniería de características para obtener buenos resultados. En otras palabras, pueden ser muy poderosos, pero requieren mucho conocimiento para implementarlos correctamente.

El análisis de asociación es relativamente ligero en los conceptos matemáticos y fácil de explicar a las personas no técnicas. Además, es una herramienta de aprendizaje no supervisada que busca patrones ocultos, por lo que existe una necesidad limitada de preparación de datos e ingeniería de

características. Es un buen comienzo para ciertos casos de exploración de datos y puede señalar el camino para una inmersión más profunda en los datos utilizando otros enfoques.

Como una ventaja adicional, la implementación de **Python** en **MLxtend** debería ser muy familiar para cualquiera que tenga experiencia con scikit-learn y pandas. Por todas estas razones, es una herramienta útil para estar familiarizado y puede ayudarnos con los problemas de análisis de datos.

Técnicamente, el análisis de la cesta de la compra es solo una aplicación del análisis de asociación. Sin embargo, es habitual utilizar análisis de asociación y análisis de la canasta de mercado indistintamente.

Las **reglas de la asociación** normalmente se escriben así:

{Pañales} -> {Cerveza}

Lo que significa que existe una fuerte relación entre los clientes que compraron pañales y también compraron cerveza en la misma transacción.

En el ejemplo anterior, el {Pañal} es el antecedente y el {Cerveza} es el consecuente. Tanto los antecedentes como los consecuentes pueden tener múltiples ítems. En otras palabras, {Pañal, Chicle} -> {Cerveza, Patatas fritas} es una regla válida.

Los componentes del análisis de asociación son los siguientes:

- El **soporte** es la frecuencia relativa con la que aparecen las reglas. En muchos casos, es posible que deseemos buscar un alto apoyo para asegurarse de que sea una relación útil. Sin embargo, puede haber casos en los que un soporte bajo sea útil si estamos tratando de encontrar relaciones "ocultas".
- La **confianza** es una medida de la fiabilidad de la regla. Una confianza de .5 en el ejemplo anterior significaría que en el 50% de los casos en que se compraron pañales y chicles, la compra también incluyó cerveza y patatas fritas. Para la recomendación del producto, una confianza del 50% puede ser perfectamente aceptable, pero en una situación médica, este nivel puede no ser lo suficientemente alto.
- **Lift** (elevación) es la relación entre el apoyo observado y el esperado si las dos reglas fueran independientes. La regla básica es que un valor de elevación cercano a 1 significa que las

reglas eran completamente independientes. Los valores de elevación > 1 son generalmente más "interesantes" y podrían ser indicativos de un patrón de regla útil.

Los datos específicos del ejemplo provienen del Repositorio de Aprendizaje Automático de la UCI y representan datos transaccionales de un minorista del Reino Unido de 2010-2011. Esto representa principalmente las ventas a mayoristas, por lo que es ligeramente diferente de los patrones de compra del consumidor, pero sigue siendo un caso de estudio útil.

Implementación del análisis de cesta

El objetivo es utilizar esta biblioteca para analizar un conjunto de datos minoristas en línea relativamente grande e intentar encontrar combinaciones de compra interesantes.

El dataset que estamos manejando es bastante grande, por lo que lanzar este código puede llevar varios minutos.

MLxtend se puede instalar usando pip, así que hay que hacerlo antes de intentar ejecutar cualquiera de los códigos a continuación.

pip install mlxtend

Una vez instalado, el código a continuación muestra cómo ponerlo en marcha.

Importamos **pandas** y **código MLxtend** y leemos los **datos**:

import pandas as p

from mlxtend.frequent_patterns import apriori

from mlxtend.frequent_patterns import association_rules

df = pd.read_excel('http://archive.ics.uci.edu/ml/machine-learning-databases/00352/Online Retail.xlsx')Descargar aquí

df.head()

Para ordenar un poco los datos, Hay que eliminar algunas de las descripciones. También eliminaremos las filas que no tienen números de factura y eliminaremos las transacciones de crédito (aquellas con números de factura que contengan C).

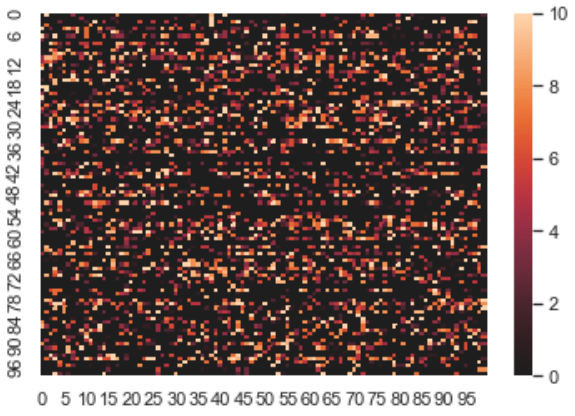
```
# Ordenar datos

df['Description'] = df['Description'].str.strip()

df.dropna(axis=0, subset=['InvoiceNo'], inplace=True)

df['InvoiceNo'] = df['InvoiceNo'].astype('str')

df = df[~df['InvoiceNo'].str.contains('C')]
```

Después de la **limpieza**, necesitamos consolidar los elementos en 1 transacción por fila con cada **producto codificado** en one hot. En aras de mantener el conjunto de datos pequeño, solo estamos trabajaremos con las ventas de Francia. Sin embargo, en el código después se compararán estos resultados con las ventas de Alemania. Sería interesante investigar más comparaciones de países.

```
basket = (df[df['Country'] =="France"]

.groupby(['InvoiceNo', 'Description'])['Quantity']

.sum().unstack().reset_index().fillna(0)

.set_index('InvoiceNo'))
```

Así es como se ven las primeras columnas:

```
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x1c8ee6fd208>
```



Hay muchos ceros en los **datos**, pero también debemos asegurarnos de que los **valores positivos** se conviertan en 1 y cualquier cosa menos que el 0 se establezca en 0. Este paso completará la codificación one hot de los datos y eliminará la columna de franqueo - postage (ya que ese cargo no es uno que deseemos explorar):

```
def encode_units(x):

if x <= 0:

return 0

if x >= 1:

return 1

basket_sets = basket.applymap(encode_units)

basket_sets.drop('POSTAGE', inplace=True, axis=1)
```

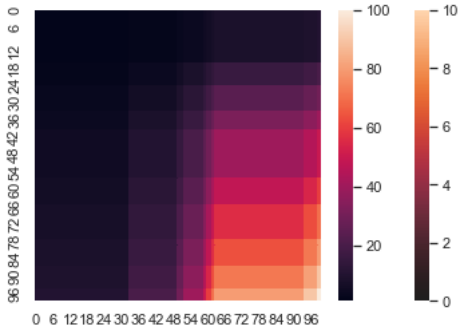
Ahora que los **datos** están **estructurados** correctamente, podemos generar conjuntos de **elementos frecuentes** que tengan un soporte de al menos el 7% (este número nos permite obtener suficientes ejemplos útiles):

```
frequent_itemsets = apriori(basket_sets, min_support=0.07, use_colnames=True)
```

El paso final es generar las reglas con su correspondiente apoyo, confianza y lift:

```
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1)

rules.head()
```

```
Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x1c8da476308>
```

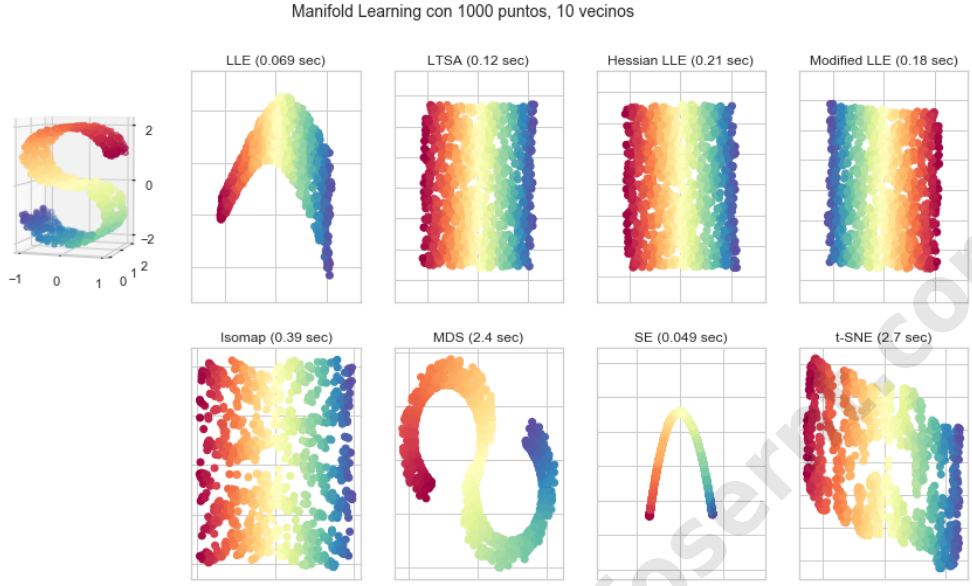


Ahora, viene la parte de interpretar los datos. Por ejemplo, podemos ver que hay bastantes reglas con un alto valor de elevación, lo que significa que ocurre con más frecuencia de lo que cabría esperar dado el número de combinaciones de transacciones y productos. También podemos ver varios datos donde la confianza es alta también. Esta parte del análisis es donde el conocimiento del dominio será útil.

Podemos filtrar el dataframe usando código pandas estándar. En este caso, buscamos un lift grande (6) y una alta confianza (.8):

```
rules[ (rules['lift'] >= 6) &

(rules['confidence'] >= 0.8) ]
```

```
LLE: 0.069 sec
LTSA: 0.12 sec
Hessian LLE: 0.21 sec
Modified LLE: 0.18 sec
Isomap: 0.39 sec
MDS: 2.4 sec
SE: 0.049 sec
t-SNE: 2.7 sec
```



Al observar las **reglas**, parece que los **despertadores verdes y rojos** se compran juntos y los vasos, servilletas y platos de papel rojo se compran juntos con una frecuencia más alta de lo que sugeriría la probabilidad general.

En este punto, es posible que deseemos ver cuánta oportunidad hay de utilizar la popularidad de un producto para impulsar las ventas de otro. Por ejemplo, podemos ver que vendemos 340 despertadores verdes, pero solo 316 despertadores rojos, así que tal vez podamos impulsar más ventas de relojes despertadores rojos a través de recomendaciones.

basket['ALARM CLOCK BAKELIKE GREEN'].sum()

340.0

basket['ALARM CLOCK BAKELIKE RED'].sum()

316.0

Lo que también es interesante es ver cómo varían las combinaciones según el país de compra. Echemos un vistazo a algunas combinaciones populares en Alemania:

```
basket2 = (df[df['Country'] =="Germany"]
.groupby(['InvoiceNo', 'Description'])['Quantity']
.sum().unstack().reset_index().fillna(0)
.set_index('InvoiceNo'))
basket_sets2 = basket2.applymap(encode_units)
basket_sets2.drop('POSTAGE', inplace=True, axis=1)
frequent_itemsets2 = apriori(basket_sets2, min_support=0.05, use_colnames=True)
rules2 = association_rules(frequent_itemsets2, metric="lift", min_threshold=1)
rules2[ (rules2['lift'] >= 4) &
(rules2['confidence'] >= 0.5)]
```

Parece que los alemanes prefieren Plasters in Tin Spaceboy y Woodland Animals.

A continuación, se lista el **código completo**.

```
import pandas as pd
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
df = pd.read_excel('http://archive.ics.uci.edu/ml/machine-learning-databases/00352/Online Retail.xlsx')
df.head()
# Ordenar datos
df['Description'] = df['Description'].str.strip()
df.dropna(axis=0, subset=['InvoiceNo'], inplace=True)
```

```
df['InvoiceNo'] = df['InvoiceNo'].astype('str')

df = df[~df['InvoiceNo'].str.contains('C')]

#Datos de Francia

basket = (df[df['Country'] =="France"]

.groupby(['InvoiceNo', 'Description'])['Quantity']

.sum().unstack().reset_index().fillna(0)

.set_index('InvoiceNo'))

#basket.head()

def encode_units(x):

if x <= 0:

return 0

if x >= 1:

return 1

basket_sets = basket.applymap(encode_units)

basket_sets.drop('POSTAGE', inplace=True, axis=1)

frequent_itemsets = apriori(basket_sets, min_support=0.07, use_colnames=True)

rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1)

#rules.head()

rules[ (rules['lift'] >= 6) &

(rules['confidence'] >= 0.8) ]

#rules.head(n=22)
```

```
basket2 = (df[df['Country'] =="Germany"]

.groupby(['InvoiceNo', 'Description'])['Quantity']

.sum().unstack().reset_index().fillna(0)

.set_index('InvoiceNo'))

basket_sets2 = basket2.applymap(encode_units)

basket_sets2.drop('POSTAGE', inplace=True, axis=1)

frequent_itemsets2 = apriori(basket_sets2, min_support=0.05, use_colnames=True)

rules2 = association_rules(frequent_itemsets2, metric="lift", min_threshold=1)

rules2[ (rules2['lift'] >= 4) &

(rules2['confidence'] >= 0.5)]

rules2.head(n=22)
```

Una aplicación específica a menudo se llama análisis de cesta de mercado. El ejemplo más conocido de análisis de cesta de mercado es el llamado caso de "cerveza y pañales". La historia básica es que un gran minorista pudo extraer sus datos de transacciones y encontrar un patrón de compra inesperado de personas que compraban cerveza y pañales para bebés al mismo tiempo.

- Verdadero.
- Falso.

Recuerda

[[[Elemento Multimedia]]]

Autoevaluación

¿Cuál es una posible aplicación de algoritmos de clasificación?

- Análisis de los sentimientos.
- Detección de peatones en la conducción de un automóvil.
- Ambas son ciertas.

Señala cuales de los siguientes algoritmos de aprendizaje no supervisados son de tipo Clustering:

- Basado en densidad.
- Basado en centroides.
- Ambos.

¿Cuál de los siguientes es un tipo de aprendizaje Manifold?

- DBSCAN.
- BIRCH.
- LLE.

¿Para qué se utiliza la librería Seaborn?

Cálculos matemáticos.

Machine Learning.

Visualización.

¿Qué algoritmo no necesita datos de forma circular para trabajar bien?

BIRCH.

Mezcla gaussiana.

K-Means.