Aprenentatge no supervisat

lloc:	Institut d'Ensenyaments a Distància de les Illes Balears	Imprès per:	Carlos Sanchez Recio
Curs:	Sistemes d'aprenentatge automàtic	Data:	dimarts, 24 de desembre 2024, 15:07
Llibre:	Aprenentatge no supervisat		

Taula de continguts

1. Clústering

En l'aprenentatge supervisat, usàvem tècniques per construir models d'aprenentatge automàtic en què les dades tenien una resposta coneguda: les etiquetes de classe estaven disponibles en les dades d'entrenament. En aquest llibre, canviarem de paradigma i explorarem l'anàlisi de clústers, una categoria de tècniques d'**aprenentatge no supervisat** que permet trobar estructures en dades de les quals no sabem l'organització correcta amb antelació. L'objectiu de la clusterització, o agrupament, és trobar grups naturals en les dades de forma que els elements de cada clúster siguin més semblants entre ells que no amb els de clústers diferents.

La clusterització, per la seva naturalesa exploratòria, és un àrea emocionant, i en aquest apartat estudiarem els conceptes següents, que ens ajudaran a organitzar les dades en estructures significatives.

- Trobar centres de semblança usant l'algorisme **k-means**.
- Usar una aproximació de baix a dalt per construir arbres de clústering jeràrquics.
- Identificar formes arbitràries d'objectes usant una aproximació basada en l'agrupació per densitat.

1.1. k-means

En aquest apartat treballarem un dels algorismes de clústering més usats, tant a la indústria com en entorns acadèmics: **k-means**. El clústering, o l'anàlisi de clústers, és una tècnica que ens permet trobar grups d'objectes semblants que estan més relacionats entre ells que no amb els elements d'altres grups. Com a exemples d'aplicacions de negoci, tenim l'agrupament de documents, música o films en diferents temes, o trobar grups de clients que comparteixen interessos semblants, en base a conductes prèvies de compra, com a fonament de motors de recomanació.

Clusterització k-means amb scikit-learn

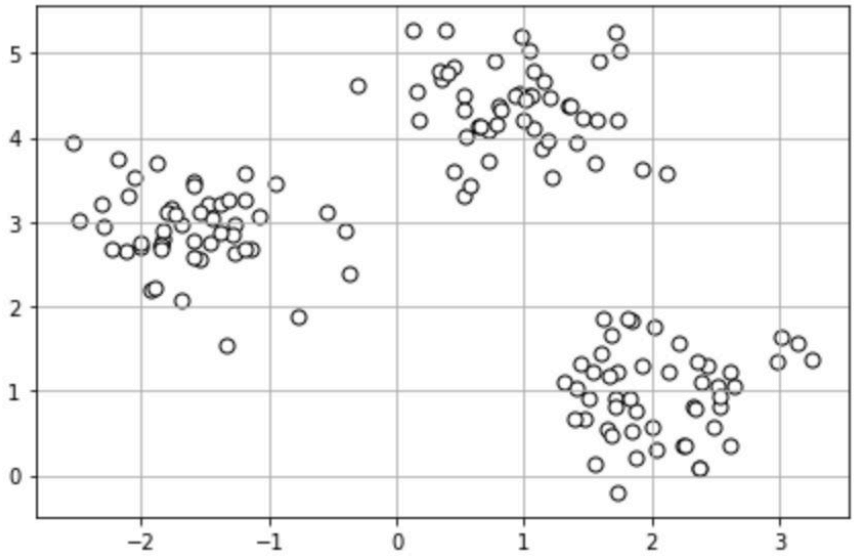
Com veurem d'aquí a poc, l'algorismes k-means (k mitjanes) és molt fàcil d'implementar, però també és molt eficient computacionalment comparat amb altres algorismes de clusterització, cosa que explica la seva popularitat. L'algorisme k-means pertany a la categoria de **clusterització basada en prototips**. Hi ha també altres tipus de clusterització, com la **clusterització jeràrquica** i la **clusterització basada en densitat**.

La clusterització basada en prototips consisteix en representar cada clúster justament per un prototip, que sol ésser o bé el **centroide** (mitjana) dels punts semblants amb característiques contínues, o el **medoide** en el cas de característiques categòriques, el punt més representatiu o el punt que minimitza la distància a tots els altres punts que pertanyen al clúster. L'algorisme k-means és molt bo per trobar grups de forma esfèrica, però un inconvenient que té és que hem de fixar a priori el nombre de clústers, k. Una tria inadequada de k pot dur a un agrupament de poca qualitat. Més endavant en aquest apartat, veurem el mètode del colze i els gràfics de silueta, tècniques útils per avaluar la qualitat d'un agrupament que ens ajudaran a determinar el nombre òptim de clústers k.

Tot i que la clusterització **k-means** es pot aplicar en espais de moltes dimensions, farem els exemples següents en dues dimensions, i això ens ajudarà a visualitzar la tècnica.

El codi següent genera 150 punts aleatoris agrupats més o manco en tres regions de més densitat, que es veuen bé mitjançant un gràfic de punts.

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=150,
                  n_features=2,
                  centers=3,
                  cluster_std=0.5,
                  shuffle=True,
                  random_state=0)

import matplotlib.pyplot as plt
plt.scatter(X[:,0],
            X[:,1],
            c='red',
            marker='o',
            edgecolor='black',
            s=50)
plt.grid()
plt.tight_layout()
plt.show()
```



En les aplicacions reals de clusterització, no tenim una referència de les categories dels exemples; de fet, si tinguéssim etiquetes de classe, ja seríem dins aprenentatge supervisat. Per tant, el nostre objectiu aquí és agrupar els exemples basant-nos en les semblances de les característiques, usant l'algorisme k-means amb els quatre passos següents.

1. Triar a l'atzar k centroides entre els exemples, com a centres de clúster inicials.
2. Assignar cada exemple al centroide més proper.
3. Moure els centroides al centre dels exemples que li han estat assignats.
4. Repetir les passes 2 i 3 fins que les assignacions a clústers no varien, fins que s'ha arribat a una tolerància predefinida o fins que s'arriba a un màxim nombre d'iteracions.

Ara, la qüestió següent és, **com mesuram la semblança entre objectes**? Podem definir semblança com l'oposat a distància. Una mesura de distància usada habitualment per clusteritzar exemples amb característiques contínues és la distància euclídia quadràtica entre dos punts x i y, en un espai *m*-dimensional.

$$d(x,y)^2 = \sum_{j=1}^{m}(x_j - y_j)^2 = ||x-y||_2^2$$

A l'equació anterior, tenim dos vectors d'entrada d'exemple x i y. L'índex j es refereix a la j-èsima dimensió d'aquests vectors. A la resta d'aquest apartat, usarem el superíndex i per indicar l'índex de l'exemple -punt de dades- i el superíndex j per indicar l'índex de clúster.

A partir d'aquesta distància, l'algorisme k-means és un problema d'optimització. Es minimitza iterativament la suma d'errors quàdratics (SSE, Squared Sum of Errors) intraclúster. Aquesta mesura també es denomina de vegades inèrcia del clúster.

$$SSE = \sum_{i=1}^{N}\sum_{j=1}^{K} w^{(i,j)}||x^{(i)} - \mu^{(j)}||_2^2$$

Aquí $\mu^{(j)}$ és el punt representatiu (centroide) del clúster j. El pes $w^{(i,j)} = 1$ si l'exemple $x^{(i)}$ és al cluster j, o 0 si no.
Ara que sabem com funciona l'algorisme k-means, apliquem-lo al nostre conjunt de dades sintètic amb la classe **KMeans** del mòdul **cluster** de scikit-learn.

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=3,
            init='random',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
y_km = km.fit_predict(X)
```

En aquest codi, fixam el nombre de clústers desitjat $k = 3$; haver d'especificar-ho a priori és una de les limitacions de k-means. També fixam `n_init = 10` per executar l'algorisme deu vegades independentment, amb deu inicialitzacions aleatòries dels centroides, i seleccionar al final el model amb SSE mínima. Amb `max_iter = 300` especificam el nombre màxim d'iteracions. La implementació de k-means de scikit-learn atura abans si convergeix abans del nombre màxim d'iteracions. És possible que l'algorisme no arribi a la convergència en una execució particular, i això pot ser problemàtic (costós computacionalment) si triam valors massa grans de `max_iter`. Una manera d'evitar problemes de convergència és triar valors grans de `tol`, la tolerància respecte dels canvis en la SSE intraclúster per considerar que s'ha arribat a la convergència. En aquest exemple, triam `tol=1e-04` ($= 0.0001$).

Un problema de k-means és que hi pot haver qualque clúster buit. A la implementació de scikit-learn, però, si un clúster queda buit s'hi assignarà el punt de dades més llunyà.

IMPORTANT

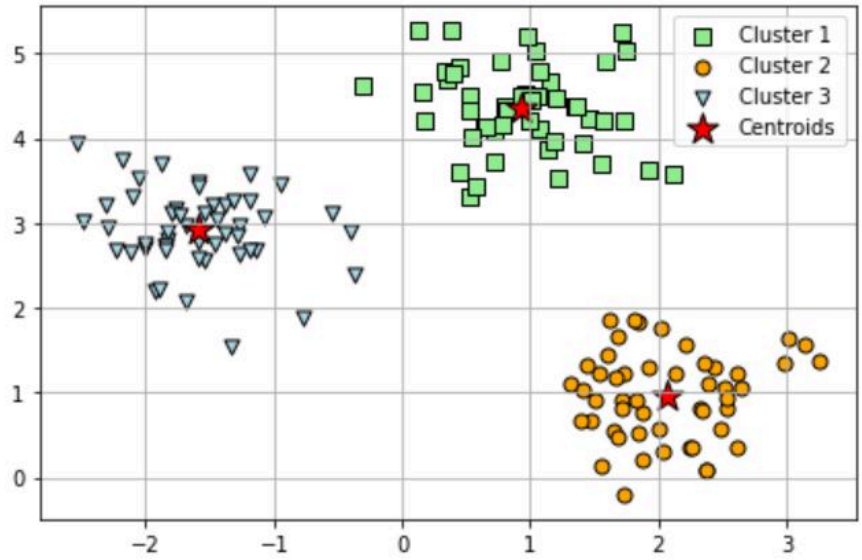
Escalat de característiques

Quan s'aplica k-means al món real amb una mètrica de distància euclidiana, hem d'assegurar-nos que les característiques estan en la mateixa escala. Per aconseguir-ho, s'aplica l'estandardització z-score (usant mitjana i desviació típica) o min-max si fa falta.

Després de predir les etiquetes de clúster, `y_km`, i de discutir alguns reptes de l'algorisme k-means, vegem els clústers que l'algorisme ha identificat en el conjunt de dades, juntament amb els centroides dels clústers. Estan emmagatzemants en l'atribut `cluster_centers_` de l'objecte ajustat `KMeans`.

```
plt.scatter(X[y_km == 0, 0],
            X[y_km == 0, 1],
            s=50, c='lightgreen',
            marker = 's', edgecolor = 'black',
            label = 'Cluster 1')
plt.scatter(X[y_km == 1, 0],
            X[y_km == 1, 1],
            s=50, c='orange',
            marker = 'o', edgecolor = 'black',
            label = 'Cluster 2')
plt.scatter(X[y_km == 2, 0],
            X[y_km == 2, 1],
            s=50, c='lightblue',
            marker = 'v', edgecolor = 'black',
            label = 'Cluster 3')
plt.scatter(km.cluster_centers_[:, 0],
            km.cluster_centers_[:, 1],
            s=250, marker='*',
            c='red', edgecolor='black',
            label='Centroids')
plt.legend(scatterpoints=1)
plt.grid()
plt.tight_layout()
plt.show()
```

Al gràfic següent, veim que l'algorisme k-means ha situat els tres centroides al centre de cada esfera, cosa que sembla raonable donades aquestes dades.



Tot i que k-means ha funcionat correctament en aquest conjunt de dades de laboratori, destaquem un inconvenient de k-means: cal especificar d'avançada el nombre de clústers k. En aplicacions reals, quin ha de ser el nombre de clústers no és tan obvi, sobretot si treballam amb dades de dimensionalitat elevada (moltes variables) que no es poden visualitzar totalment. Altres propietats de k-means són que les clústers no es poden solapar i no són jeràrquics. També se suposa que en cada clúster hi ha almanco un element.

1.2. clusterització jeràrquica

En aquest apartat estudiarem una alternativa a la clusterització basada en prototips: la **clusterització jeràrquica**. Un avantatge de la clusterització jeràrquica és que permet la representació de **dendrogrames** (visualitzacions d'una clusterització jeràrquica binària), que poden ajudar a interpretar a la interpretació dels resultats a través de la creació de taxonomies significatives. Un altre avantatge de la clusterització jeràrquica és que no cal especificar per avançat el nombre de clústers.

Les dues formes principals de clusterització jeràrquica són l'**aglomerant** i la **divisiva**. En la clusterització jeràrquica divisiva començam amb un sol clúster que inclou tot el conjunt de dades i el dividim iterativament en clústers més petits fins que cada clúster conté un sol exemple. En aquesta secció ens centrarem en la clusterització aglomerativa, en què cada inicialment cada clúster conté un sol exemple i es van agrupant els parells de clústers més propers fins que només n'hi ha un.

1.3. Agrupament de clústers de baix a dalt

Els dos algorismes estàndard per a la clusterització jeràrquica aglomerativa són el **lligament simple** i el **lligament complet**.

En el lligament simple, calculam les distàncies entre els membres més semblants de cada parell de clústers i unim els dos clústers per als quals la distància entre els membres més semblants és mínima.

En el lligament complet, la diferència és que feim servir com a criteri per a unir clústers la distància entre els membres més dissemblants. Ajuntam els clústers de distància mínima entre els seus membres més dissemblants.

AMPLIACIÓ

Uns altres algorismes per a la clusterització jeràrquica aglomerativa són el **lligament promig** i el **lligament de Ward**.

En el **lligament promig**, ajuntam els clústers de distància mínima tenint en compte la suma de les distàncies entre tots els parells de membres.

En el **lligament de Ward**, s'ajunten els dos clústers que duen a un increment mínim de la suma d'errors quadràtics intraclúster.

A continuació ens centrarem en la clusterització aglomerativa usant el lligament complet. Usarem un procediment iteratiu format per les passes següents.

1. Calcular la matriu de distàncies de tots els exemples
2. Representar cada punt de dades com un clúster per a ell tot sol
3. Fusionar els dos clústers més propers basant-nos en la distància entre els membres més dissemblants, més distants
4. Recalcular la matriu de distàncies
5. Repetir les passes 2-4 fins que només quedi un únic clúster

Tot seguit veurem com calcular la matriu de distàncies (passa 1). Però abans haurem de generar unes dades aleatòries per poder-hi treballar. Les fileres representen diferents observacions (ID 0-4), i les columnes són diferents característiques (*features*) X, Y i Z dels exemples.

```
import pandas as pd
import numpy as np
np.random.seed(123)
variables = ['X','Y','Z']
labels = ['ID_0','ID_1','ID_2','ID_3','ID_4']
X = np.random.random_sample([5, 3])*10
df = pd.DataFrame(X, columns=variables, index=labels)
df
```

Després d'executar el codi, obtenim un dataframe amb els exemples aleatoris que hem generat.

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

1.4. Clusterització jeràrquica sobre una matriu de distàncies

Calcularem la matriu de distàncies per a l'entrada de l'algorisme de clusterització, amb la funció pdist del submòdul de SciPy spatial.distance

```
from scipy.spatial.distance import pdist, squareform
row_dist = pd.DataFrame(squareform(
                    pdist(df, metric='euclidean')),
                    columns = labels, index = labels)
row_dist
```

Amb aquest codi, hem calculat la distància euclidiana entre tots els parells d'exemples del conjunt de dades, en base a les característiques X, Y i Z.

Hem donat la matriu de distàncies condensada (que retorna pdist) com a entrada a la funció squareform per crear una matriu simètrica de les distàncies per parelles.

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

A continuació, aplicam l'aglomeració per lligament complet amb la funció linkage del submòdul de SciPy cluster.hierarchy, que retorna la matriu de lligament.

Això sí, abans de fer servir la funció linkage, consultem la documentació de la funció.

```
from scipy.cluster.hierarchy import linkage
help(linkage)
```

```
Parameters
----------
y : ndarray
    A condensed distance matrix. A condensed distance matrix
    is a flat array containing the upper triangular of the distance matrix.
    This is the form that ``pdist`` returns. Alternatively, a collection of
    :math:`m` observation vectors in :math:`n` dimensions may be passed as
    an :math:`m` by :math:`n` array. All elements of the condensed distance
    matrix must be finite, i.e., no NaNs or infs.
method : str, optional
    The linkage algorithm to use. See the ``Linkage Methods`` section below
    for full descriptions.
metric : str or function, optional
    The distance metric to use in the case that y is a collection of
    observation vectors; ignored otherwise. See the ``pdist``
    function for a list of valid distance metrics. A custom distance
    function can also be used.
optimal_ordering : bool, optional
    If True, the linkage matrix will be reordered so that the distance
    between successive leaves is minimal. This results in a more intuitive
    tree structure when the data are visualized. defaults to False, because
    this algorithm can be slow, particularly on large datasets [2]_. See
    also the `optimal_leaf_ordering` function.

    .. versionadded:: 1.0.0

Returns
-------
Z : ndarray
    The hierarchical clustering encoded as a linkage matrix.
```

En base a aquesta documentació, veim que hem d'usar una matriu de distància condensada (triangular superior) de la funció pdist com a atribut d'entrada. Si no, també podríem donar el vector de dades inicials i usar la mètrica 'euclidean' com a argument de la funció linkage. En aquest cas, no hauríem d'usar la matriu de distàncies

squareform que hem definit abans, ja que això donaria resultats erronis. En resum, els tres possibles escenaris són:

Incorrecte: Usar la matriu de distància squareform com en el codi següent dóna resultat incorrecte.

```
row_clusters = linkage (row_dist,
                        method='complete',
                        metric='euclidean')
row_clusters
```

Correcte: Si usam la matriu de distàncies condensada com a l'exemple següent el resultat és correcte.

```
row_clusters = linkage(pdist(df, metric='euclidean'), method='complete')
```

Correcte: Si usam la matriu d'exemples d'entrada (matriu de disseny) també obtenim un resultat correcte igual que en el cas anterior.

```
row_clusters = linkage(df.values, method='complete', metric='euclidean')
```

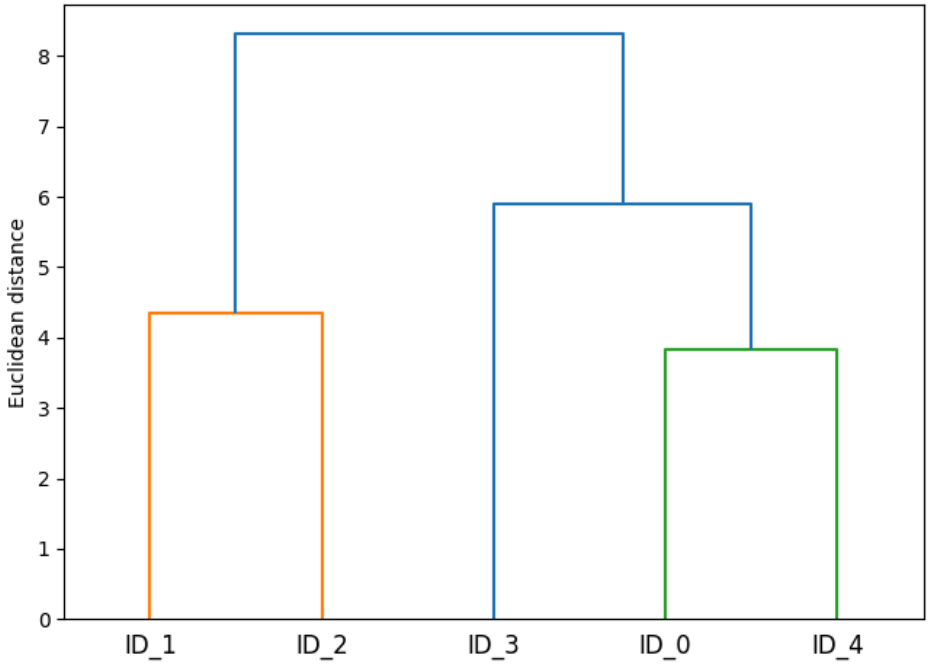
Vegem els resultats de la clusterització transformant el resultat en un DataFrame de pandas.

```
pd.DataFrame(row_clusters,
             columns=['row label 1',
                      'row label 2',
                      'distance',
                      'no. of items in clust.'],
             index=['cluster %d' % (i+1) for i in range(row_clusters.shape[0])])
```

Com veim a continuació, la matriu de lligament consisteix en diferents fileres en què cada filera representa una fusió. La primera i la segona columna són els membres més diferents de cada clúster, i la tercera columna indica la distància entre aquests dos membres. La quarta columna dona el nombre de membres de cada clúster.

	row label 1	row label 2	distance	no. of items in clust.
cluster 1	0.0	4.0	3.835396	2.0
cluster 2	1.0	2.0	4.347073	2.0
cluster 3	3.0	5.0	5.899885	3.0
cluster 4	6.0	7.0	8.316594	5.0

Ara que hem calculat la matriu de lligament, podem visualitzar els resultats en forma de dendrograma.



Aquest dendrograma resumeix els diferents clústers que s'han format durant la clusterització jeràrquica aglomerativa. Per exemple, hi veim que els exemples ID_0 i ID_4, seguits de ID_1 i ID_2, són els més semblants d'acord amb la matriu de distància euclidiana.

1.5. Dendrograma amb un mapa de calor

En aplicacions pràctiques, sovint es combinen els dendrogrames de clusterització jeràrquica en combinació amb un **mapa de calor**. Així es representen els valors individuals en el vector de dades o la matriu d'exemples amb un codi de color. En aquest apartat veurem com connectar un dendrograma a un gràfic de mapa de calor i com ordenar les fileres del mapa de calor convenientment.

El procediment és una mica enrevessat, de forma que el veurem pas a pas.

1. Cream un nou objecte figure i hi definim la posició de l'eix x, la posició de l'eix y, amplada i alçada del dendrograma amb l'atribut add_axes. A més, rotam el dendrograma 90 graus en sentit antihorari.

```
fig = plt.figure(figsize=(8,8), facecolor='white')
axd = fig.add_axes([0.09, 0.1, 0.2, 0.6])
row_dg = dendrogram(row_clusters, orientation='left')
```

2. A continuació, reordenam les dades del nostre DataFrame iniical d'acord amb les etiquetes de clusterització que tenim a l'objecte dendrogram, un diccionari Python, a través de la clau leaves.

```
df_rowclust = df.iloc[row_dg['leaves'][::-1]]
```

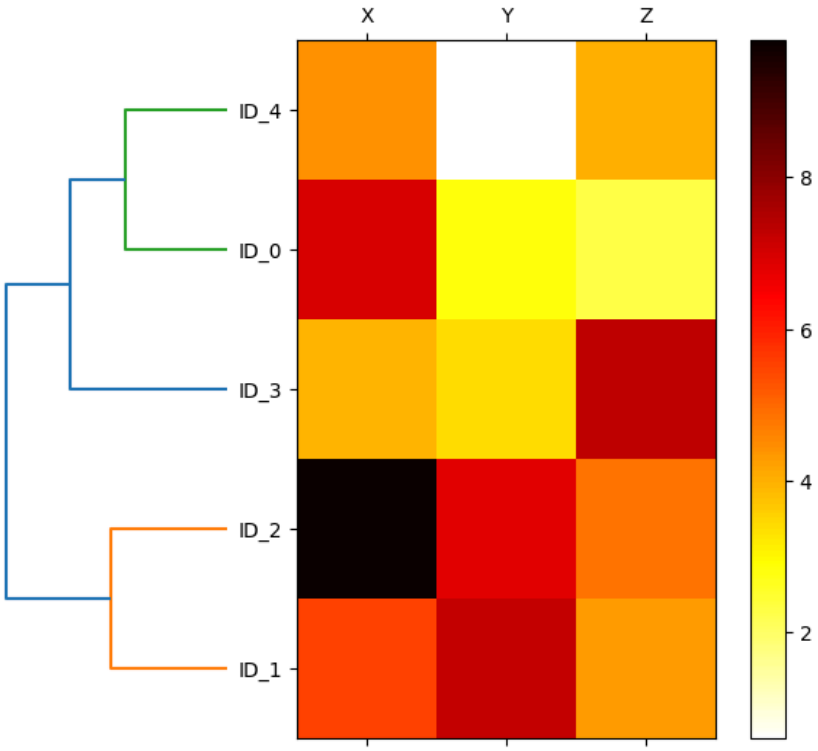
3. Ara construïm el mapa de calor a partir del DataFrame reordenat i el situam vora el dendrograma.

```
axm = fig.add_axes([0.23, 0.1, 0.6, 0.6])
cax = axm.matshow(df_rowclust,
                  interpolation='nearest',
                  cmap='hot_r')
```

4. Finalment, modificam l'aspecte del dendrograma llevant les marques dels eixos i amagant les puntes dels eixos. També afegim una barra de color i assignam els noms de característica i registre a les etiquetes dels eixos x i y.

```
axd.set_xticks([])
axd.set_yticks([])
for i in axd.spines.values():
  i.set_visible(False)
fig.colorbar(cax)
axm.set_xticklabels(['']+list(df_rowclust.columns))
axm.set_yticklabels(['']+list(df_rowclust.index))
plt.show()
```

Després d'aplicar aquests passos, es veu el mapa de calor amb el dendrograma adjunt.



L'ordre de les fileres al mapa de calor reflecteix la clusterització dels exemples al dendrograma. Més complet que un simple dendrograma, els valors en color de cada característica i cada exemple de les dades ens donen un resum vistós del conjunt de dades.

1.6. Clusterització aglomerativa amb scikit-learn

Hem vist fins ara com realitzar la clusterització jeràrquica aglomerativa amb la biblioteca **SciPy**. Tanmateix, també hi ha una implementació **AgglomerativeClustering** a **scikit-learn**, que permet fixar el nombre de clústers que es volen retornar. Això és útil quan es vol podar l'arbre. Fixant el paràmetre n_cluster al valor 3, ara clusteritzarem els exemples d'entrada en tres grups, usant el mateix lligament complet basat en la distància euclidiana, com abans.

```
from sklearn.cluster import AgglomerativeClustering

ac = AgglomerativeClustering(n_clusters=3,
                             metric='euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print('Cluster labels: %s' % labels)
```

```
Cluster labels: [1 0 0 2 1]
```

Mirant les etiquetes de clúster predites, veim que el primer i el cinquè exemples (ID_0 i ID_4) s'assignen a un clúster (etiqueta 1) i que els exemples ID_1 i ID_2 s'assignen a un segon clúster (etiqueta 2). En general els resultats són coherents amb els que hem observat al dendrograma. Notem, però, que ID_3 és més semblant a ID_4 i ID_0 que no a ID_1 i ID_2, com es veu al dendrograma; això no s'aprecia als resultats de scikit-learn. Tornem a executar AgglomerativeClustering amb n_cluster=2.

```
ac = AgglomerativeClustering(n_clusters = 2,
                             metric = 'euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print('Cluster labels: %s' % labels)
```

```
Cluster labels: [0 1 1 0 0]
```

Com es pot veure, en aquesta jerarquia podada, l'etiqueta ID_3 s'assigna al mateix clúster que ID_0 i ID_4, com esperàvem.

1.7. DBSCAN

Vegem una tercera forma de clusterització en aquest apartat, basada en la densitat. DBSCAN són les sigles de **density-based spatial clustering of applications with noise**. En aquest cas no es fa cap suposició sobre la forma dels clústers, que en k-means se suposa esfèrica, ni es fa una partició del dataset en jerarques amb un punt de tall manual. DBSCAN assigna les etiquetes dels clústers a partir de les regions de punts denses. La densitat al voltant d'un punt es defineix com el nombre de punts que hi té dins un radi especificat, epsilon.

En l'algorisme DBSCAN, s'assigna una de les tres etiquetes especials a cada punt de dades d'acord amb el criteri següent.

Un punt es considera nuclear (core point) si com a mínim hi ha un nombre de punts veïns (MinPts) dins un radi especificat epsilon.

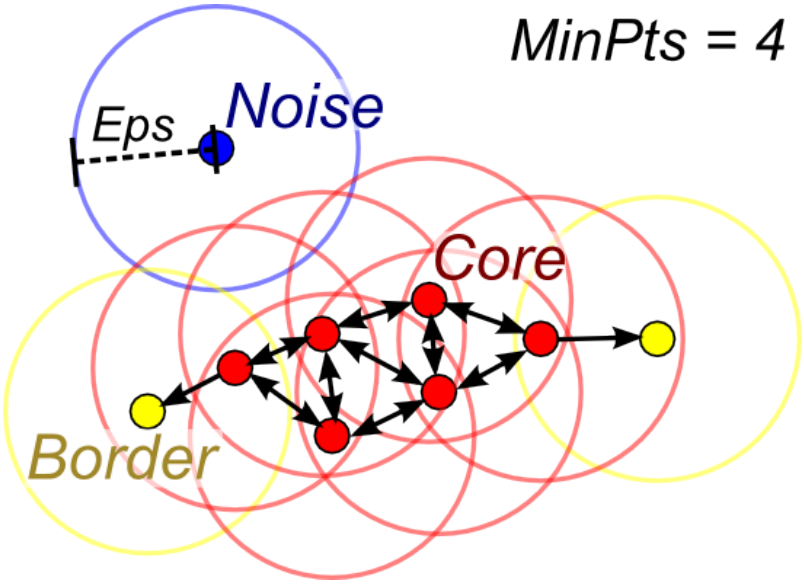
Un punt es considera de frontera (border point) si té menys de MinPts veïnats dins un radi epsiolon, però és dins el radi d'un punt nuclear.

La resta de punts, que no són ni de nucli ni de frontera, es consideren punts de soroll.

Després d'etiquetar tots els punts com a nucli, frontera o soroll, l'algorisme DBSCAN consisteix en les dues passes següents.

1. Formar un clúster per a cada punt nucli o conjunt de punts nucli. Els punts nuclears es connecten si la seva distància no supera epsilon.
2. Assignar cada punt frontera al clúster que correspon al seu nucli.

Abans de passar a la implementació, vegem una il·lustració dels punts nucli, frontera i soroll.



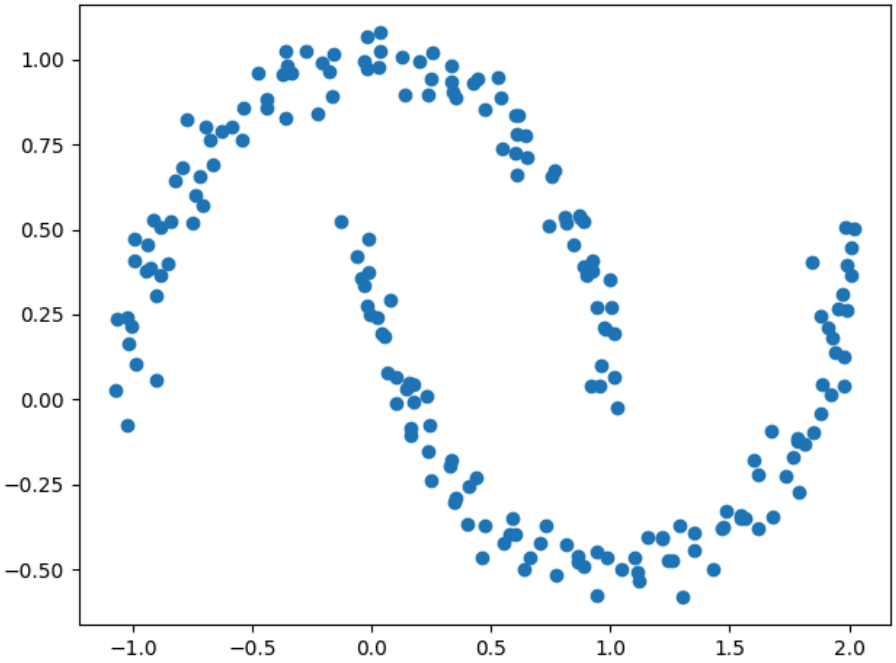
Imatge: *DBSCAN: punts nucli, frontera i soroll*

Una dels principals avantatges de DBSCAN és que no suposa que els clústers tenguin una forma esfèrica com en k-means. A més, una altra diferència respecte de la clusterització k-means o jeràrquica és que no necessàriament assigna tots els punts a un clúster, sinó que pot eliminar els que es considerin soroll perquè són fora de les zones de densitat elevada.

Farem un exemple il·lustratiu a partir d'un conjunt de dades amb estructures amb forma de mitja lluna, per comparar els tres mètodes de clusterització que hem vist.

```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt

X, y = make_moons(n_samples=200,
                  noise=0.05,
                  random_state=0)

plt.scatter(X[:, 0],X[:, 1])
plt.tight_layout()
plt.show()
```

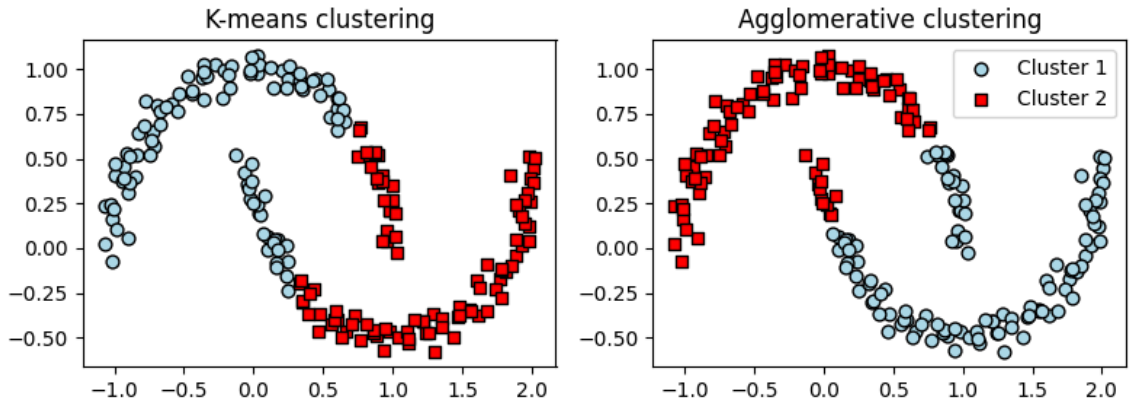
Hi ha dos grups de 100 punts cadascun, amb forma de mitja lluna.



Començarem usant l'algorisme de clusterització k-means i el de lligament complet per veure si poden agrupar aquestes estructures amb forma de mitja lluna.

```
from sklearn.cluster import KMeans, AgglomerativeClustering

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8,3))
km = KMeans(n_clusters=2, random_state=0, n_init='auto')
y_km = km.fit_predict(X)

ax1.scatter(X[y_km == 0, 0],
            X[y_km == 0, 1],
            c ='lightblue',
            edgecolor='black',
            marker='o',
            s=40,
            label='cluster 1')

ax1.scatter(X[y_km == 1, 0],
            X[y_km == 1, 1],
            c='red',
            edgecolor='black',
            marker='s',
            s=40,
            label='cluster 2')

ax1.set_title('K-means clustering')
ac = AgglomerativeClustering(n_clusters=2,
                             metric='euclidean',
                             linkage='complete')

y_ac = ac.fit_predict(X)

ax2.scatter(X[y_ac == 0, 0],
            X[y_ac == 0, 1],
            c='lightblue',
            edgecolor='black',
            marker='o',
            s=40,
            label='Cluster 1')

ax2.scatter(X[y_ac == 1, 0],
            X[y_ac == 1, 1],
            c='red',
            edgecolor='black',
            marker='s',
            s=40,
            label='Cluster 2')

ax2.set_title('Agglomerative clustering')
plt.legend()
plt.tight_layout()
plt.show()
```

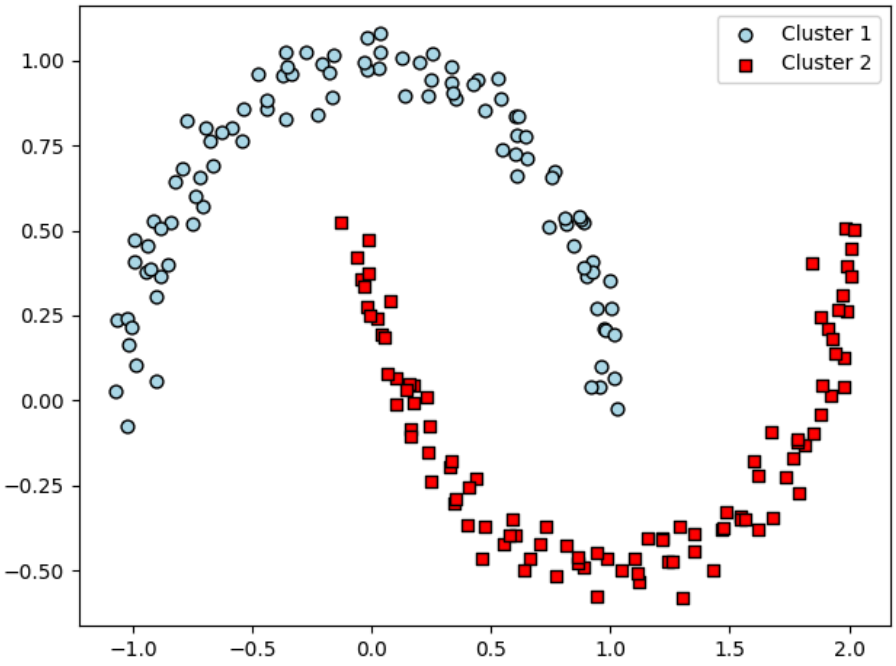
Veim que cap dels dos algorismes no agrupa correctament els punts.



Finalment, vegem el resultat d'aplicar l'algorisme DBSCAN a les dades amb forma de mitja lluna.

```
from sklearn.cluster import DBSCAN
db = DBSCAN(eps = 0.2,
            min_samples=5,
            metric='euclidean')

y_db = db.fit_predict(X)
plt.scatter(X[y_db == 0, 0],
            X[y_db == 0, 1],
            c='lightblue',
            edgecolor='black',
            marker='o',
            s=40,
            label='Cluster 1')
plt.scatter(X[y_db == 1, 0],
            X[y_db == 1, 1],
            c='red',
            edgecolor='black',
            marker='s',
            s=40,
            label='Cluster 2')
plt.legend()
plt.tight_layout()
plt.show()
```

Aquest algorisme sí que se'n surt, com podem observar al gràfic.



Tanmateix, hem de notar alguns dels desavantatges de DBSCAN. Amb un nombre gran de característiques al dataset, suposant un nombre fix d'exemples d'entrenament, l'efecte negatiu de la **maledicció de la dimensionalitat** (*curse of dimensionality*) creix. Això és especialment problemàtic si usam la mètrica de la distància euclidiana. De tota manera, aquest problema també apareix en la clusterització k-means o jeràrquica, no és exclusiu de DBSCAN. A més, tenim dos hiperparàmetres de DBSCAN (MinPts i epsilon) que hem d'optimitzar per obtenir bons resultats. Trobar una bona combinació de MinPts i epsilon pot ser problemàtic si les diferències de densitat són relativament grans.

Observem que a la pràctica no sempre és obvi quin algorisme funcionarà més bé amb un determinat conjunt de dades, sobretot si les dades són en múltiples dimensions que fan la visualització difícil o impossible.

1.8. Clústering basat en models

Del llibre AIMA.

Gaussian Mixture Models

2. Reducció de dimensionalitat amb PCA

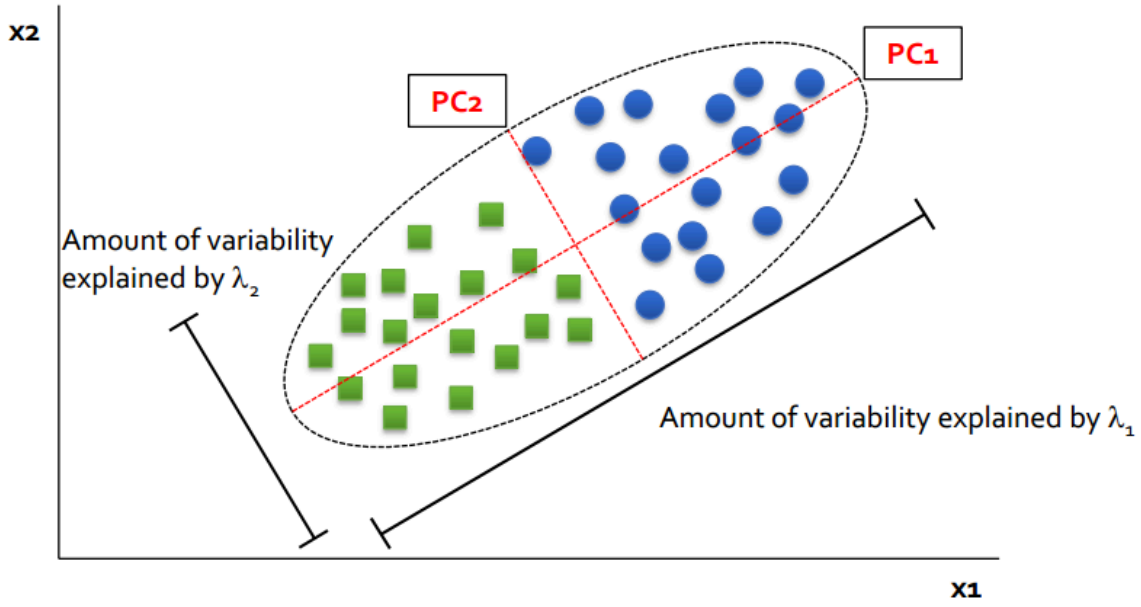
Per reduir la dimensionalitat d'un conjunt de dades, podem fer servir diverses tècniques d'**extracció de característiques** (*feature extraction*), transformant o projectant les dades en un nou espai de característiques.

En aquest context de reducció de la dimensionalitat, l'extracció de característiques es pot entendre com una forma de compressió de dades que té l'objectiu de mantenir la major part de la informació rellevant. A la pràctica, l'extracció de característiques no només s'usa per reduir l'espai d'emmagatzematge o l'eficiència computacional de l'algorisme d'aprenentatge, sinó que pot millorar la capacitat predictiva, sobretot si treballam amb models no regularitzats.

2.1. Etapes de l'anàlisi PCA

A continuació veurem l'anàlisi de components principals, **PCA** (*Principal Component Analysis*). És una tècnica que s'usa en diferents àmbits, per a l'extracció de característiques i la reducció de dimensionalitat. Altres aplicacions de PCA inclouen l'anàlisi de dades exploratòria i la reducció de soroll en l'operació en el mercat de valors, l'anàlisi genètica en bioinformàtica.

PCA ens ajuda a identificar patrons a les dades a partir de la correlació entre característiques. Bàsicament, PCA troba les direccions de variància màxima en les dades en moltes dimensions i projecta les dades en un nou subespai de les mateixes o més poques dimensions que l'original. Els eixos ortogonals (components principals) del nou subespai es poden interpretar com les direccions de màxima variància amb la restricció que els nous eixos de característiques siguin ortogonals, com mostra la figura següent.



Imatge: *Anàlisi de components principals*

A la figura anterior, $x1$ i $x2$ són els eixos de característiques originals, i $PC1$ i $PC2$ són les components principals.

Quan usam PCA per a la reducció de dimensionalitat, construïm una matriu de transformació de dimensions d x k, anomenada W, que permet transformar un vector, x, les característiques d'un exemple d'entrenament, en un nou subespai de característiques de k dimensions amb $k < d$.

Com a resultat de transformar les dades que originalment tenen d dimensions en aquest nou espai de k dimensions (típicament $k << d$) la primera component principal té la variància més gran possible. Les següents components principals tendran la més gran variància possible donada la restricció que aquestes components siguin mútuament ortogonals (incorrelades). Les direccions PCA són molt sensibles a l'escalat, i hem d'estandarditzar les característiques abans de realitzar PCA si les característiques es mesuren en diferents escales i volem donar la mateixa importància a totes les característiques.

Resumim el procés en les pases següents.

1. Estandarditzar el conjunt de dades d-dimensional
2. Construir la matriu de covariància
3. Descomposar la matriu de covariància en els seus autovectors i autovalors
4. Ordenar els autovalors en ordre decreixent per prioritzar els autovectors corresponents
5. Seleccionar els k autovectors d'autovalor més alt, on k és la dimensió desitjada del nou subespai de característiques
6. Construir una matriu de projecció, W, a partir dels k autovectors d'autovalor més gran

7. Transformar el conjunt d'entrada d-dimensional, X, usant la matriu de projecció, W, per obtenir el nou subespai de característiques k-dimensional

Als apartats següents, implementarem PCA pas a pas, usant Python. Després veurem com realitzar PCA amb scikit-learn.

2.2. Extracció de les components principals

En aquest apartat treballarem les primeres quatre passes de l'anàlisi PCA

1. Estandardització de les dades
2. Construcció de la matriu de covariància
3. Obtenció dels autovalors i autovectors de la matriu de covariància
4. Ordenació dels autovalors i autovectors

Començarem carregant el conjunt de dades Wine.

```
import pandas as pd

df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data', header=None)
```

A continuació, dividirem les dades en dos subconjunts d'entrenament i prova, en proporció 70% / 30%. Els estandarditzam a variància unitat.

```
from sklearn.model_selection import train_test_split
X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y,
random_state=0)

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

Ara podem construir la matriu de covariància. La matriu de covariància és quadrada i de dimensions dxd. Emmagatzema els valors de la covariància entre cada parell de característiques. La covariància entre dues característiques, al nivell de la població, es pot calcular amb la següent expressió.

$$\sigma_{jk} = \frac{1}{n-1} \sum_{k=1}^{n} (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k)$$

Aquí, μ_j i μ_k són les mitjanes mostrals de les característiques j i k, respectivament. Observem que les mitjanes mostrals són zero si estandarditzam el conjunt de dades. Una covariància positiva entre dues característiques indica que les característiques creixen o decreixen alhora, mentre que una covariància negativa indica que varien en direccions oposades: quan una creix l'altra decreix, i a la inversa.

Els autovectors de la matriu de covariància representen les components principals, les direccions de variància màxima, mentre que els autovalors corresponents indiquen la seva magnitud. En el cas del conjunt de dades Wine, obtenim 13 autovectors i autovalors a partir de la matriu de covariància de dimensions 13x13.

A continuació, per al tercer pas, obtenim els autoparells (autovalor i autovector) de la matriu de covariància. Els autovectors d'una matriu tenen la propietat que quan es multipliquen per la matriu, el resultat que s'obté és senzillament una versió escalada del vector. Això vol dir que la seva direcció no varia quan s'hi aplica la transformació representada per la matriu. En notació algebraica, això s'escriu de la forma següent.

$$\Sigma v = \lambda v$$

En l'expressió anterior, λ, l'autovalor és un escalar. El càlcul a mà dels autovalors i autovectors és llarg i complicat. Per això, usarem la funció linalg.eig del NumPy per obtenir els autoparells de la matriu de covariància de Wine.

```
import numpy as np

cov_mat = np.cov(X_train_std.T)
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
print('\nEigenvalues \n%s' % eigen_vals)
```

```
Eigenvalues
[4.84274532 2.41602459 1.54845825 0.96120438 0.84166161 0.6620634
 0.51828472 0.34650377 0.3131368  0.10754642 0.21357215 0.15362835
 0.1808613 ]
```

Amb la funció numpy.cov, hem calculat la matriu de covariància del conjunt de dades d'entrenamet estandarditzat. Amb la funció linalg.eig hem realitzat l'autodescomposició. El resultat ha estat un vector de 13 autovalors (eigen_vals) i els autovectors corresponents disposats en columna en una matriu de dimensió 13x13 (eigen_vecs).

2.3. Variància total i variància explicada

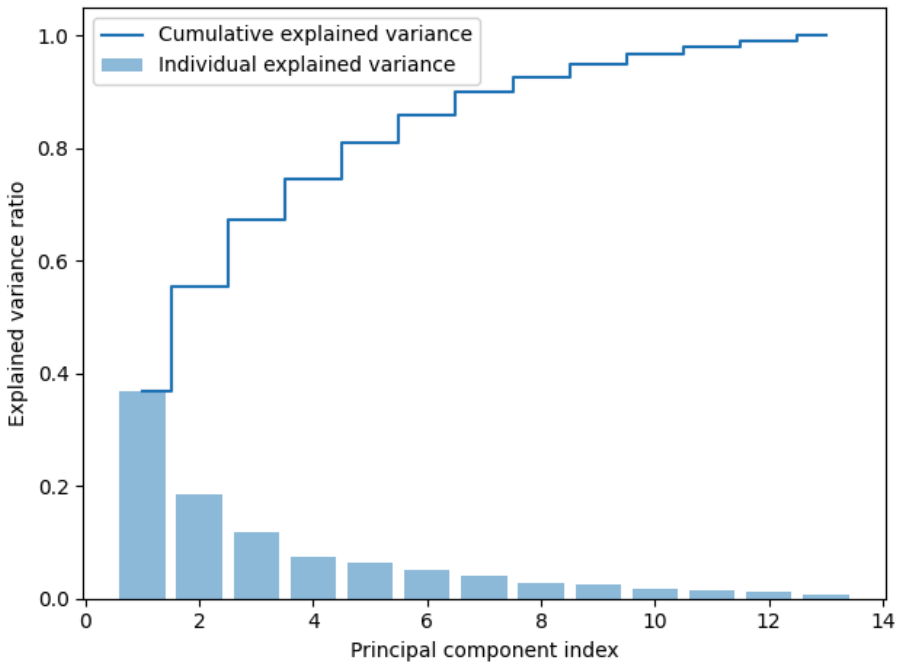
Com que l'objectiu d'aquest procediment PCA és reduir la dimensionalitat del conjunt de dades comprimint-lo en un nou subespai de característiques, només seleccionam el subconjunt d'autovectors (components principals) que contenen la major part de la informació, representada per la variància. Els autovalors defineixen la magnitud dels autovectors en la matriu de covariància, per això hem d'ordenar els autovectors segons la magnitud del seu autovalor associat. Ens interessen els k autovectors més importants, els que tenen autovalor més gran. Però abans de seleccionar aquests k autovectors, vegem les **raons de variància explicada** dels autovalors.
La raó de variància explicada EVR d'un autovalor λ_j es defineix com el quocient entre aquest autovalor i la suma de tots els autovalors de la matriu.

$$EVR = \frac{\lambda_j}{\sum_{k=1}^{d} \lambda_k}$$

Amb la funció cumsum de NumPy, podem calcular la suma acumulada de les variàncies explicades, que després representarem amb la funció step de Matplotlib.

```
total = sum(eigen_vals)
var_exp = [(i/total) for i in sorted (eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)

import matplotlib.pyplot as plt
plt.bar(range(1,14), var_exp, alpha=0.5, align='center', label='Individual explained
variance')
plt.step(range(1,14), cum_var_exp, where='mid', label='Cumulative explained variance')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component index')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

El gràfic resultant indica que la primera component tota sola ja explica aproximadament el 40% de la variància total. També veim que les dues primeres components expliquen gairebé un 60% de la variància del conjunt de dades.



Recordem que PCA és un mètode d'aprenentatge no supervisat. Això implica que s'ignora la informació de les etiquetes de classe. La variància ens dona una idea de l'amplitud del rang de valors al llarg d'un eix de

característiques.

2.4. Transformació de característiques

Després de les quatre passes de l'apartat anterior, seguim amb les tres darreres passes de la transformació.

5. Selecció de k autovectors, que corresponen als k autovalors més grans, on k és la dimensionalitat del nou espai de característiques. Habitualment $k << d$.
6. Construcció de la matriu de projecció W a partir dels k autovectors més representatius.
7. Transformació del conjunt de dades de dimensió d, usant la matriu de projecció W, per obtenir el nou subespai de característiques de dimensió k.

Començam ordenant els autoparells per ordre decreixent dels autovectors.

```
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
               for i in range(len(eigen_vals))]
eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

A continuació, prenem els dos autovectors que corresponen als dos autovalors més grans, per capturar en aquest cas el 60% de la variància del conjunt. En prenem dos per poder-ho visualitzar fàcilment al pla amb un diagrama bidimensional. A la pràctica, el nombre d'autovalors s'ha de determinar com un compromís entre l'eficiència computacional i el rendiment del classificador.

```
w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
               eigen_pairs[1][1][:, np.newaxis]))

print('Matrix W:\n', w)
```

```
Matrix W:
 [[-0.13724218  0.50303478]
 [ 0.24724326  0.16487119]
 [-0.02545159  0.24456476]
 [ 0.20694508 -0.11352904]
 [-0.15436582  0.28974518]
 [-0.39376952  0.05080104]
 [-0.41735106 -0.02287338]
 [ 0.30572896  0.09048885]
 [-0.30668347  0.00835233]
 [ 0.07554066  0.54977581]
 [-0.32613263 -0.20716433]
 [-0.36861022 -0.24902536]
 [-0.29669651  0.38022942]]
```

Amb el codi anterior, hem creat una matriu de projecció de dimensions 13x2, a partir dels dos primers autovectors.

Utilitzant la matriu de projecció W, ara podem transformar un exemple x sobre el subespai PCA (les components principals 1 i 2) i obtenim x', un exemple bidimensional que consisteix en dues noves característiques.

```
X_train_std[0].dot(w)
```

```
array([2.38299011, 0.45458499])
```

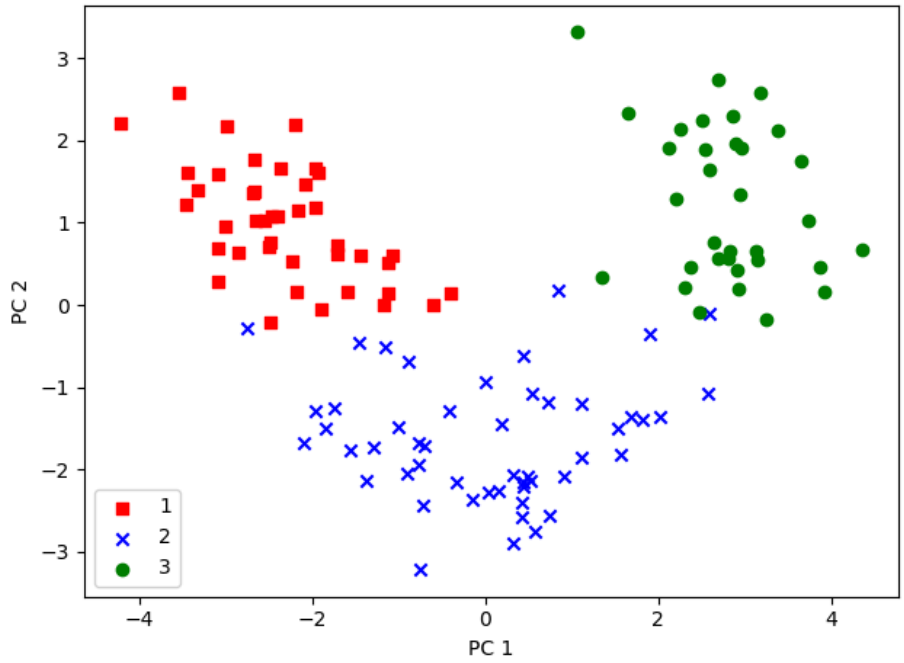
Igualment, podem transformar tot el conjunt de dimensions 124x13 on dues components principals calculant el producte.

```
X_train_pca = X_train_std.dot(w)
```

Finalment, vegem en un gràfic bidimensional com queda el conjunt transformat, ara emmagatzemat en una matriu 124x2.

```
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for l, c, m in zip(np.unique(y_train), colors, markers):
  plt.scatter(X_train_pca[y_train==l, 0],
              X_train_pca[y_train==l, 1],
              c=c, label=l, marker=m)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```

Podem observar que les dades estan més escampades al llarg de l'eix x, la primera component principal, que no a l'eix y, la segona component. Això és coherent amb les variàncies explicades per cada component, que hem vist abans. Després d'aquesta transformació no supervisada, un classificador (supervisat) podria distingir bé les classes.



Subratllem una vegada més que el procediment d'anàlisi PCA no utilitza les etiquetes de classe.

2.5. PCA amb scikit-learn

La implementació detallada que hem vist ara és útil per entendre bé com funciona PCA per dins. Ara veurem com utilitzar la classe PCA implementada a scikit-learn.

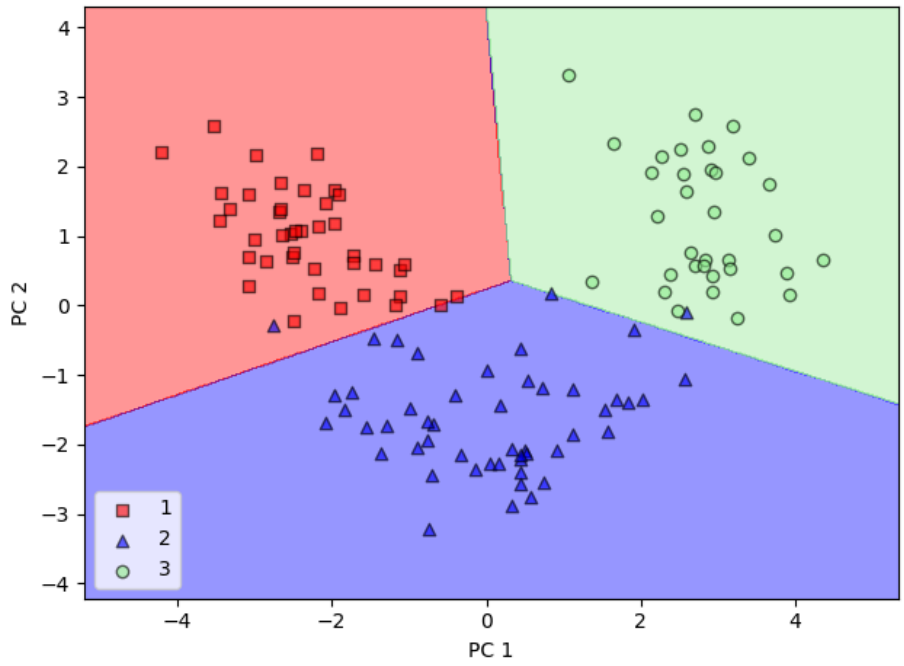
La classe PCA és una de les classes de transformació que ofereix scikit-learn. Primer ajustam el model amb les dades d'entrenament i després aplicam la transformació als dos subconjunts, d'entrenament i test, usant els mateixos paràmetres. A continuació, usarem la classe PCA de scikit-learn el conjunt de dades Wine, classificarem els exemples transformats amb regressió logística (encara que aquesta passa queda fora d'aquest lliurament d'aprenentatge no supervisat) i visualitzarem les regions de decisió amb la funció plot_decision_regions.

```
from matplotlib.colors import ListedColormap

def plot_decision_regions (X, y, classifier, resolution=0.02):
  markers = ('s', '^', 'o', 'v', '>')
  colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
  cmap = ListedColormap(colors[:len(np.unique(y))])

  x1_min, x1_max = X[:, 0].min() -1, X[:, 0].max() + 1
  x2_min, x2_max = X[:, 1].min() -1, X[:, 1].max() + 1
  xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                         np.arange(x2_min, x2_max, resolution))

  Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)

  Z = Z.reshape(xx1.shape)

  plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
  plt.xlim(xx1.min(), xx1.max())
  plt.ylim(xx2.min(), xx2.max())

  for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y==cl, 0],
                y=X[y==cl, 1],
                alpha=0.6,
                color=cmap(idx),
                edgecolor='black',
                marker=markers[idx],
                label=cl)
```

Per comoditat, es pot posar aquest codi plot_decision_region en un fitxer separat del directori de treball, per exemple, plot_decision_regions_script.py, i importar-lo des de la sessió de treball actual.

```
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
lr = LogisticRegression(multi_class='ovr',
                        random_state=1,
                        solver='lbfgs')

X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)

lr.fit(X_train_pca, y_train)
plot_decision_regions(X_train_pca, y_train, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```

Després d'executar aquest codi, podem veure les regions de decisió de les dades d'entrenament reduïdes a dos eixos de components principals.

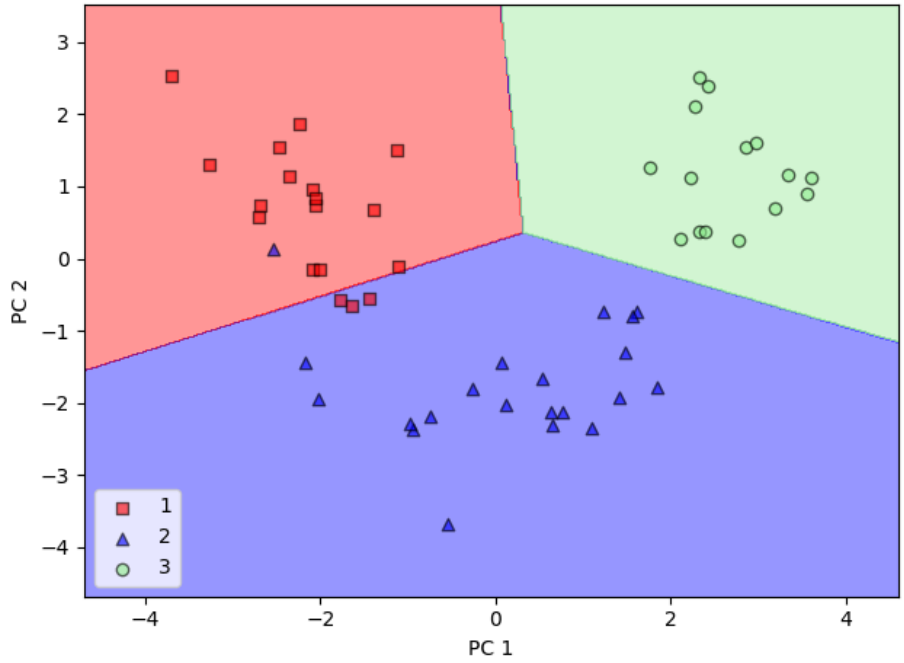


Quan comparam les projeccions PCA a través de scikit-learn amb les de la nostra pròpia implementació, podem trobar que els gràfics siguin una imatge reflectida l'un de l'altre. Això no vol dir que hi hagi cap errada en cap de les dues implementacions, sinó que segons l'algorisme de resolució dels autovalors el seu signe pot ser positiu o negatiu.

Podríem corregir aquesta diferència, si volguéssim, multiplicant les dades per -1. Per tenir una visió més completa, vegem les regions de decisió de la regressió logística sobre el conjunt de dades transformat, per veure si pot separar bé les classes.

```
plot_decision_regions(X_test_pca, y_test, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```

Al gràfic podem observar que el classificador de regressió logística funcionarà prou bé sobre aquest subespai de característiques bidimensional i només classifica erròniament alguns exemples del conjunt de prova.



Si ens interessa conèixer les raons de variància explicada de les diferents components principals, podem inicialitzar la classe PCA amb el paràmetre n_components fixat a None, de forma que es mantenguin totes les components. Llavors podrem accedir a l'atribut explained_variance_ratio.

```
pca = PCA(n_components=None)
X_train_pca = pca.fit_transform(X_train_std)
pca.explained_variance_ratio_
```

```
array([0.36951469, 0.18434927, 0.11815159, 0.07334252, 0.06422108,
       0.05051724, 0.03954654, 0.02643918, 0.02389319, 0.01629614,
       0.01380021, 0.01172226, 0.00820609])
```

Quan indicam n_components=None a la inicialització de la classe PCA, en comptes de realitzar la reducció de dimensionalitat es retornen totes les components principals ordenades.

3. Detecció d'anomalies

La detecció de valors atípics té l'objectiu de separar un nucli d'observacions regulars d'algunes de contaminants, anomenades *outliers* en anglès, dades atípiques o observacions atípiques.

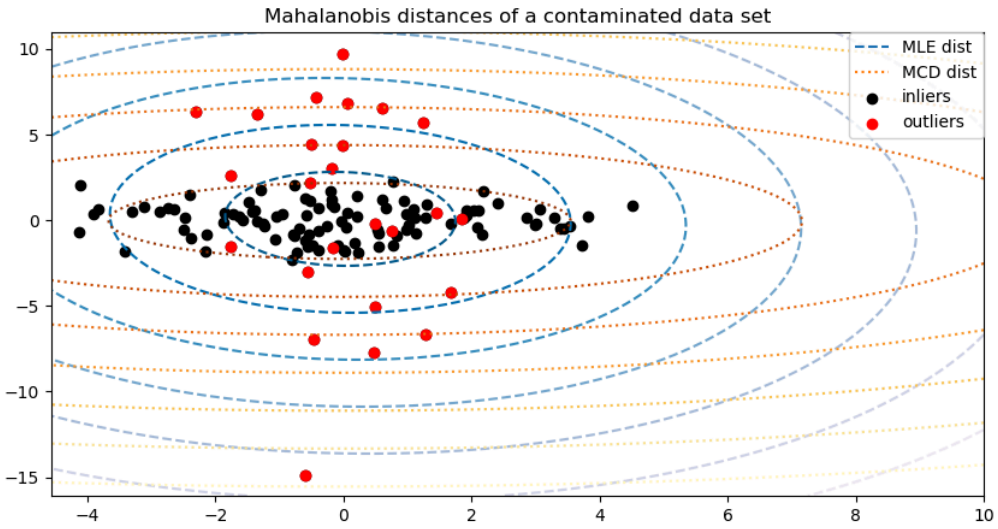
En aquesta situació no tenim un conjunt de dades net que representi la població d'observacions regulars que es pugui utilitzar per entrenar cap eina.

3.1. Ajust d'una el·lipse

Una manera habitual de realitzar la detecció de valors atípics és suposar que les dades regulars provenen d'una distribució coneguda (per exemple, les dades estan distribuïdes de forma gaussiana). A partir d'aquesta hipòtesi, generalment intentam definir la "forma" de les dades i podem definir les observacions atípiques com a observacions que es troben prou lluny de la forma d'ajust.

Scikit-learn proporciona un objecte covariance.EllipticEnvelope que ajusta una estimació de covariància robusta a les dades i, per tant, s'ajusta una el·lipse als punts de dades centrals, ignorant els punts que queden fora del mode central.

Per exemple, assumint que les dades típiques estan distribuïdes de forma gaussiana, estimarà la ubicació i la covariància interna d'una manera robusta (és a dir, sense tenir en compte la influència dels valors atípics). Les distàncies de Mahalanobis obtingudes d'aquesta estimació s'utilitzen per obtenir una mesura de la perifèria.



https://scikit-learn.org/stable/modules/outlier_detection.html#fitting-an-elliptic-envelope

3.2. Isolation Forest

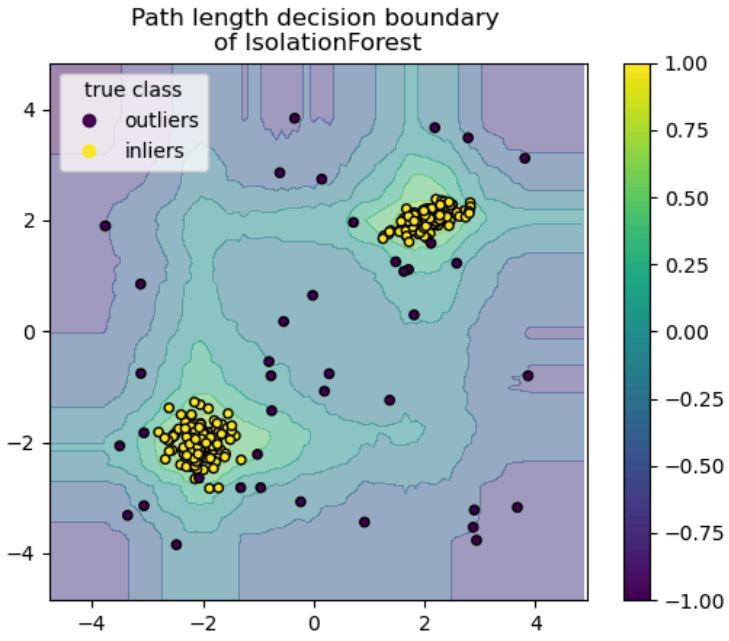
Una manera eficient de realitzar la detecció de valors atípics en conjunts de dades d'alta dimensió és utilitzar boscos aleatoris. L'ensemble.IsolationForest "aïlla" les observacions seleccionant aleatòriament una característica i després seleccionant aleatòriament un valor dividit entre els valors màxim i mínim de la característica seleccionada.

Com que la partició recursiva es pot representar mitjançant una estructura d'arbre, el nombre de divisions necessàries per aïllar una mostra és equivalent a la longitud del camí des del node arrel fins al node final.

Aquesta longitud del camí, mitjana sobre un bosc d'arbres tan aleatoris, és una mesura de la normalitat i la nostra funció de decisió.

La partició aleatòria produeix camins notablement més curts per a anomalies. Per tant, quan un bosc d'arbres aleatoris produeix col·lectivament camins més curts per a mostres particulars, és molt probable que siguin anomalies.

La implementació de ensemble.IsolationForest es basa en un conjunt de tree.ExtraTreeRegressor.



https://scikit-learn.org/stable/modules/outlier_detection.html#isolation-forest

3.3. Local Outlier Factor

Una altra manera eficient de realitzar la detecció de valors atípics en conjunts de dades de dimensions moderadament altes és utilitzar l'algorisme Local Outlier Factor (LOF).

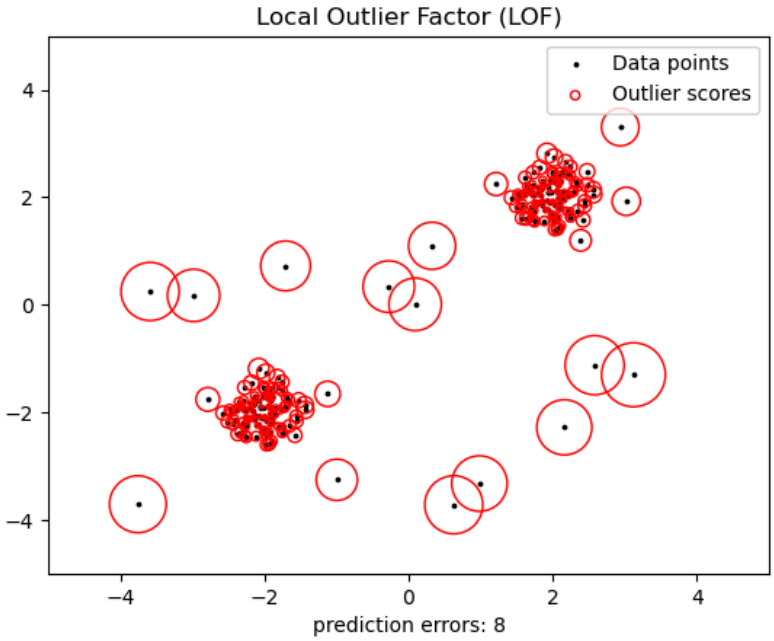
L'algorisme de neighbors.LocalOutlierFactor (LOF) calcula una puntuació (anomenada factor local outlier) que reflecteix el grau d'anormalitat de les observacions. Mesura la desviació de la densitat local d'un punt de dades donat respecte als seus veïns. La idea és detectar les mostres que tinguin una densitat substancialment inferior a la dels seus veïns.

A la pràctica, la densitat local s'obté dels k-veïns més propers. La puntuació LOF d'una observació és igual a la relació entre la densitat local mitjana dels seus k-veïns més propers i la seva pròpia densitat local: s'espera que una instància normal tengui una densitat local similar a la dels seus veïns, mentre que les dades anormals s'espera que tenguin una densitat local molt menor.

El nombre k de veïns considerats (paràmetre n_neighbors) es tria normalment 1) més gran que el nombre mínim d'objectes que ha de contenir un clúster, de manera que altres objectes poden ser atípics locals en relació amb aquest clúster, i 2) més petit que el màxim nombre d'objectes propers que potencialment poden ser atípics locals. A la pràctica, aquesta informació generalment no està disponible, i prendre n_neighbors=20 sembla que funciona bé en general. Quan la proporció de valors atípics és alta (és a dir, superior al 10 %, com a l'exemple següent), n_neighbors hauria de ser més gran (n_neighbors=35 a l'exemple següent).

La força de l'algorisme LOF és que té en compte les propietats locals i globals dels conjunts de dades: pot funcionar bé fins i tot en conjunts de dades on hi ha mostres anormals que tenen diferents densitats subjacents. La qüestió no és com d'aïllada està la mostra, sinó com d'aïllada està respecte a l'entorn que l'envolta.

Quan aplicam LOF per a la detecció de valors atípics, no hi ha mètodes predict, decision_function i score_samples, sinó només un mètode fit_predict. Les puntuacions d'anormalitat de les mostres d'entrenament són accessibles mitjançant l'atribut negative_outlier_factor_. S'ha de tenir en compte que predict, decision_function i score_samples es poden utilitzar en dades noves no vistes quan s'aplica LOF per a la detecció de novetats, és a dir, quan el paràmetre de novetat s'estableix en True, però el resultat de predict pot diferir del de fit_predict.



https://scikit-learn.org/stable/modules/outlier_detection.html#local-outlier-factor