Módulo 5.2 Métodos estándar en clasificación.

Autor:

Por Prof. Alberto Fernández Hilario

Profesor Catedrático de Universidad de Granada. Instituto Andaluz Interuniversitario en Data Science and Computational Intelligence (DasCI)

Breves Instrucciones

Recordatorio: Introducción a NoteBook

El cuaderno de *Jupyter* (Python) es un enfoque que combina bloques de texto (como éste) junto con bloques o celdas de código. La gran ventaja de este tipo de celdas es su interactividad, ya que pueden ser ejecutadas para comprobar los resultados directamente sobre las mismas.

Muy importante: el orden de las instrucciones (bloques de código) es fundamental, por lo que cada celda de este cuaderno debe ser ejecutada secuencialmente. En caso de omitir alguna, puede que el programa lance un error (se mostrará un bloque salida con un mensaje en inglés de color rojo), así que se deberá comenzar desde el principio en caso de duda. Para hacer este paso más sencillo, se puede acceder al menú “Entorno de Ejecución” y pulsar sobre “Ejecutar anteriores”.

¡Ánimo!

Haga clic en el botón "play" en la parte izquierda de cada celda de código. Las líneas que comienzan con un hashtag (#) son comentarios y no afectan a la ejecución del programa.

También puede pinchar sobre cada celda y hacer "*ctrl+enter*" (*cmd+enter* en Mac).

Cuando se ejecute el primero de los bloques, aparecerá el siguiente mensaje:

"*Advertencia: Este cuaderno no lo ha creado Google.*

El creador de este cuaderno es <autor>@go.ugr.es. Puede que solicite acceso a tus datos almacenados en Google o que lea datos y credenciales de otras sesiones. Revisa el código fuente antes de ejecutar este cuaderno. Si tienes alguna pregunta, ponte en contacto con el creador de este cuaderno enviando un correo electrónico a <autor>@go.ugr.es."

No se preocupe, deberá confiar en el contenido del cuaderno (Notebook) y pulsar en "*Ejecutar de todos modos*". Todo el código se ejecuta en un servidor de cálculo externo y no afectará en absoluto a su equipo informático. No se pedirá ningún tipo de información o credencial, y por tanto podrá seguir con el curso de forma segura.

Cada vez que ejecute un bloque, verá la salida justo debajo del mismo. La información suele ser siempre la relativa a la última instrucción, junto con todos los `print()` (orden para imprimir) que haya en el código.

ÍNDICE

En este *notebook*:

1. Se introducirán los paradigmas de clasificación de "caja blanca".
2. Se describirán los métodos de clasificación basados en modelos lineales.
3. Se estudiará el algoritmo de clasificación basado en el vecino más cercano.
4. Se presentará el algoritmo para obtener árboles de decisión para clasificación.
5. Se analizarán las ventajas e inconvenientes de todos los clasificadores anteriores.
6. Se mostrará cómo utilizar todos estos clasificadores desde Python con *Scikit-Learn*.
7. Se mostrarán ejemplos gráficos de las fronteras de decisión de cada modelo, para comprender sus principales diferencias en cuanto a su funcionamiento.

Contenidos:

1. INTRODUCCIÓN

En esta primera sección, se realizará una introducción a los paradigmas de clasificación en general, y a los específicos que se describen en este apartado del curso. A continuación, se cargarán los datos del caso de estudio de cáncer de melanoma con los que se viene trabajando hasta ahora.

1.1 Paradigmas de clasificación

En la cápsula 1 de este Módulo se destacó que existen diferentes tipos de funciones discriminantes. Esto permite enumerar a su vez distintos paradigmas o modelos de clasificación: "*modelos de caja blanca*" y "*modelos de caja negra*". Este apartado del curso se centrará principalmente en el primer tipo, ya que son la "primera línea de ataque" cuando se quiere resolver un problema de clasificación.

La razón principal es que estos modelos se obtienen de forma relativamente rápida y simple, y son en general altamente interpretables. Esto significa que se pueden examinar las componentes del modelo e identificar cuáles son las variables clave utilizadas para realizar la división entre las clases. De este modo, permiten al usuario comprender si éstas tienen un sentido con respecto al problema que se está estudiando, por ejemplo desde el punto de vista biológico.

Entre los diferentes algoritmos de clasificación destacados en este grupo, se introducirán los basados en regresión lineal y logística, el vecino más cercano, y los árboles de decisión.

1.2 Cargar los datos del problema

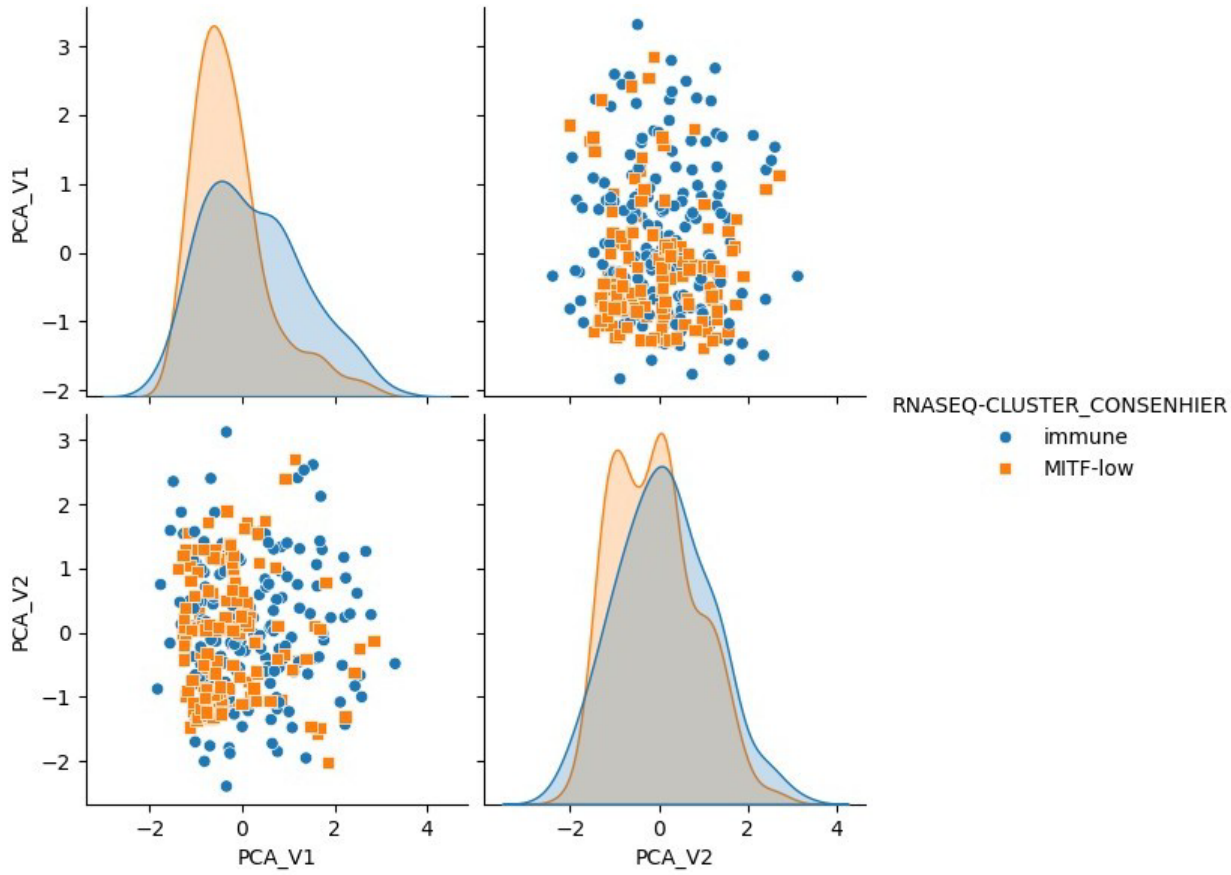
Con el objetivo de comprobar el comportamiento de los diferentes algoritmos de clasificación, se comienza incorporando los datos del caso de estudio que sirve como hilo conductor del presente curso. La notación o código utilizado es exactamente el mismo que en actividades anteriores.

```
import pandas as pd

#Cargamos los datos ómicos de la matriz de expresión desde un fichero
compartido en Google Drive
gene_exp_inmune = pd.read_csv('https://drive.google.com/uc?
id=1PYzEIdmnfjOnBpPDIFBE9hL1Lkj_OBCk',index_col=0)
#Cargamos la variable clínica correspondiente a las etiquetas "inmune"
vs. "MITF-low"
clinical_info_inmune = pd.read_csv('https://drive.google.com/uc?
id=1hHQfcvrFa5Jds-9tW_X4sHjKpYKdii9s',index_col=0)

X, y = gene_exp_inmune, clinical_info_inmune

#Imprimimos las 5 primeras muestras del conjunto de datos para
comprobar que se ha cargado correctamente
X.head()
```

{"type":"dataframe","variable_name":"X"}

Adicionalmente, se va a proceder a transformar el conjunto de datos a solamente dos dimensiones (dos variable de entrada) utilizando las componentes principales (PCA) que ya se introdujo en el Módulo 2. De este modo, se podrá representar de manera muy sencilla las fronteras de clasificación obtenidas por cada técnicas de clasificación.

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

#Transformamos el conjunto de datos inicial para que esté representado
por solo 2 variables
n_componentes = 2
pca = PCA(n_components=n_componentes)
X_2D = pca.fit_transform(X)

#Se transforma el rango de cada variable a [0, 1]
st = StandardScaler()
X_2D = st.fit_transform(X_2D)

#Transformamos el conjunto resultando a una estructura data_frame
(pandas) para facilitar la representación gráfica
df = pd.DataFrame(X_2D, columns = ['PCA_V1','PCA_V2'])
df = df.join(y)

#We show on a scatterplot the new 2D set, together with the density
functions
sns.pairplot(df, hue='RNASEQ-CLUSTER_CONSENHIER', markers=["o", "s"],
height=3);
```



Por simplicidad, para los ejemplos incluidos en este NoteBook se utilizará una validación tipo "*hold-out*" por defecto. Para más detalles consúltese el Módulo 3 sobre Aprendizaje Supervisado.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=42)

y_int = pd.get_dummies(y).iloc[:,0]
X_2D_train, X_2D_test, y_2D_train, y_2D_test = train_test_split(X_2D,
y_int, random_state=42)

print("Numero de instancias en entrenamiento: {}; y test:
{}".format(len(X_train),len(X_test)))
```

```
Numero de instancias en entrenamiento: 252; y test: 84
```

2. MODELOS SIMPLES: REGRESIÓN LINEAL Y LOGÍSTICA

En el Módulo 4 sobre Aprendizaje Supervisado: Regresión se estudió que se pueden construir funciones que aproximen los valores de salida reales para un conjunto de datos. Esta misma idea se puede aplicar directamente a clasificación, tomando en consideración que ahora los valores

de salida no están en un rango real, si no que pertenecen a un conjunto *n-ario* (binario para dos clases).

En primer lugar, se introducirá el modelo más sencillo posible basado en una regresión lineal simple, es decir, un hiperplano de separación (una "recta" que divide las instancias). A continuación, extenderemos esta idea hacia la denominada como *regresión logística*.

2.1 Modelo de regresión lineal

Se describirán las características principales de este modelo, y cómo se puede utilizar mediante Scikit-Learn.

2.1.1. Introducción al modelo lineal

De acuerdo a lo comentado al comienzo de esta sección, para adaptar el formato de aprendizaje de *regresión lineal* a clasificación, bastará con aprender o ajustar los coeficientes de un hiperplano intentando aproximar la salida a los valores por defecto {0, 1}, siguiendo el mismo esquema de minimizar la suma residual de cuadrados. En este caso, se está teniendo en cuenta que se trabaja con un problema de clasificación binario, donde la primera clase se identificará con el valor "0", mientras que la segunda tendrá el valor "1".

Antes de continuar, destacar que los hiperplanos son simples "cortes con una recta" para la clasificación de las instancias, es decir, una frontera de decisión. De esta forma, las instancias (o puntos de datos) que se encuentren a cada lado del hiperplano ("recta") serán predichas como una clase distinta. Como es lógico, la dimensión del hiperplano depende del número de variables de entrada. Si éste es 2, entonces el hiperplano es una sencilla línea recta. Cuando el número de variables es 3, entonces el hiperplano se convierte en un plano bidimensional. Manejar más de 3 dimensiones se hace complicado para un usuario humano.

La fórmula obtenida que divide el espacio de entrada en dos partes será la siguiente:

\begin{equation} \hat{y}(x,w) = w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n \end{equation}

donde $w=(w_1, \ldots, w_n)$ serán los coeficientes asociados a cada variable de entrada, y w_0 el término independiente. En el caso trivial para un problema de una única dimensión, (una variable x), se buscaría la clásica recta $y=a \cdot x+b$.

2.1.2 Implementación en Scikit-Learn

Para aplicar la regresión lineal en tareas de clasificación, la sintaxis de *Scikit-Learn* es equivalente al caso de regresión. En este caso de ejemplo, se muestran los coeficientes calculados para cada variable, almacenados en la variable miembro `coef_`, y el término independiente en `intercept_`.

```
from sklearn import linear_model

lm = linear_model.LinearRegression()

#Aquí hacemos un pequeño "truco" que es transformar la salida
categórica a valores {0,1}
#Este paso es necsario para aplicar el modelo de "regresión" que
espera una salida de tipo real.
y_train_int = pd.get_dummies(y_train).iloc[:,0]

lm.fit(X_train, y_train_int)
print(lm.intercept_)
lm.coef_
```

```
0.3371977981689106
```

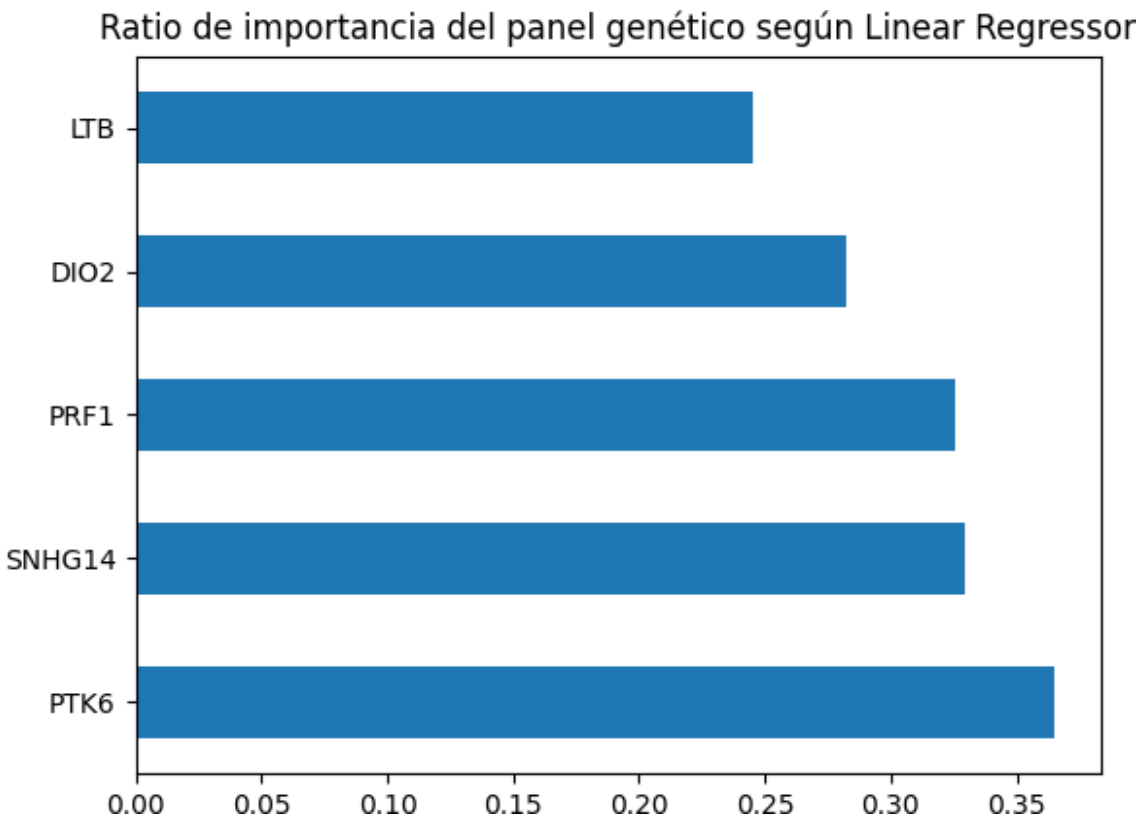
```
array([ 0.00062339, -0.01640629,  0.05765623, -0.00365438, -
0.03219924,
        0.02831667,  0.02246995,  0.02406873,  0.04154468,
0.0113776 ,
        0.03335884,  0.02299525, -0.00375194,  0.04292671,
0.01228339,
       -0.01640771,  0.03062444, -0.00302422, -0.02929467,
0.00676286,
       -0.01947947, -0.02999738,  0.02491126, -0.0079843 , -
0.00937148,
        0.00456585,  0.00615687, -0.01706812, -0.0042064 , -
```

```
0.03262455,
       -0.03213231,  0.00450434, -0.05782194, -0.02456257,
0.0250594 ,
       -0.00637356,  0.04990764, -0.00675726, -0.04263797, -
0.07777568,
        0.03133627,  0.01415696, -0.07683601, -0.08607171, -
0.01192452,
        0.02302473,  0.06651956, -0.01050754,  0.04608665, -
0.00776793])
```

Del mismo modo que sucedía en las tareas de regresión (Módulo 4), el valor absoluto de cada coeficiente indica la importancia de dicha variable en la aproximación de la salida. De este modo, podemos interpretar cuáles son las propiedades de los datos que mayor influencia tienen en la distinción de las clases.

En el siguiente cuadro, se muestra un gráfico resumen de la importancia de las variables (subconjunto del perfil genético de los 5 más importantes) en base al valor de los coeficientes de la función discriminante.

```
import numpy as np
from sklearn.preprocessing import normalize

importancia = np.abs(lm.coef_) #transformamos a una lista uni-
dimensional
#Normalizamos los valores:
importancia_norm = normalize(importancia[:,np.newaxis],
axis=0).ravel()

#Se representan las 5 más importantes según valor absoluto
(pd.Series(importancia_norm,
index=X_train.columns).nlargest(5).plot(kind='barh'))
plt.title("Ratio de importancia del panel genético según Linear
Regressor")
plt.show()
```



A continuación, se examinará la calidad de predicciones generadas de acuerdo a la medida estándar de accuracy (porcentaje de acierto). En primer lugar, observamos que la salida obtenida por el modelo no es un valor discreto, si no que es un valor real dado que estamos usando una fórmula de regresión.

No obstante, como se indicó anteriormente se va a considerar cada clase como un valor entero. En este caso de estudio de clasificación binaria, la clase "*MITF-Low*" será clase 0, mientras que "*immune*" será clase 1. Así, se procederá con un umbral de corte igual a 0.5 para determinar la clase final.

```
from sklearn.metrics import accuracy_score

#Volvemos a transformar la salida a un valor {0,1}
y_test_int = pd.get_dummies(y_test).iloc[:,0]
y_pred = lm.predict(X_test)

print("Valores de predicción originales (10 primeros):")
print(y_pred[:10])
print()

print("Valores de predicción redondeados (10 primeros):")

y_pred[y_pred >= 0.5] = 1
y_pred[y_pred < 0.5] = 0
```

```
print(y_pred[:10])
print()

acc_score = accuracy_score(y_test_int, y_pred)
print("Accuracy obtenido:",acc_score)
```

```
Valores de predicción originales (10 primeros):
[ 1.29485689 -0.09064197  1.20110728  1.05823504  0.32454412
1.00993704
  0.65879706  0.6314562   0.2538342   0.45818503]

Valores de predicción redondeados (10 primeros):
[1. 0. 1. 1. 0. 1. 1. 1. 0. 0.]

Accuracy obtenido: 0.7142857142857143
```

2.2 Modelo de Regresión Logística

2.2.1 Introducción a la regresión logística

A pesar de los buenos resultados mostrados, la regresión lineal resulta limitada en muchos casos. Por este motivo, en las tareas de Ciencia de Datos se prefiere utilizar una técnica conocida como *regresión logística*. A pesar de su nombre, es un modelo lineal de clasificación más que de regresión.

En este caso, la función de salida que se desea aproximar sería la siguiente:

\begin{equation} \hat{y}(w,x) = \frac{1}{1+e^{ w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n}} \ end{equation}

La ventaja de esta aproximación, sobre la regresión lineal, es que crea una separación más suave entre los valores de la variable de salida (clases):

2.2.2 Implementación en Scikit-Learn y principales parámetros

A continuación, se indican los pasos para construir un clasificador de este tipo en Scikit-Learn. Obsérvese que se sigue exactamente el mismo esquema que el visto anteriormente para `LinearRegressor`; sin embargo, en este caso al tratarse de un clasificador, la salida sí está en el conjunto `{0, 1}`.

Existen diferentes parámetros que se pueden configurar, aunque se recomienda dejar los valores por defecto. En cualquier caso, a continuación se enumeran los más importantes:

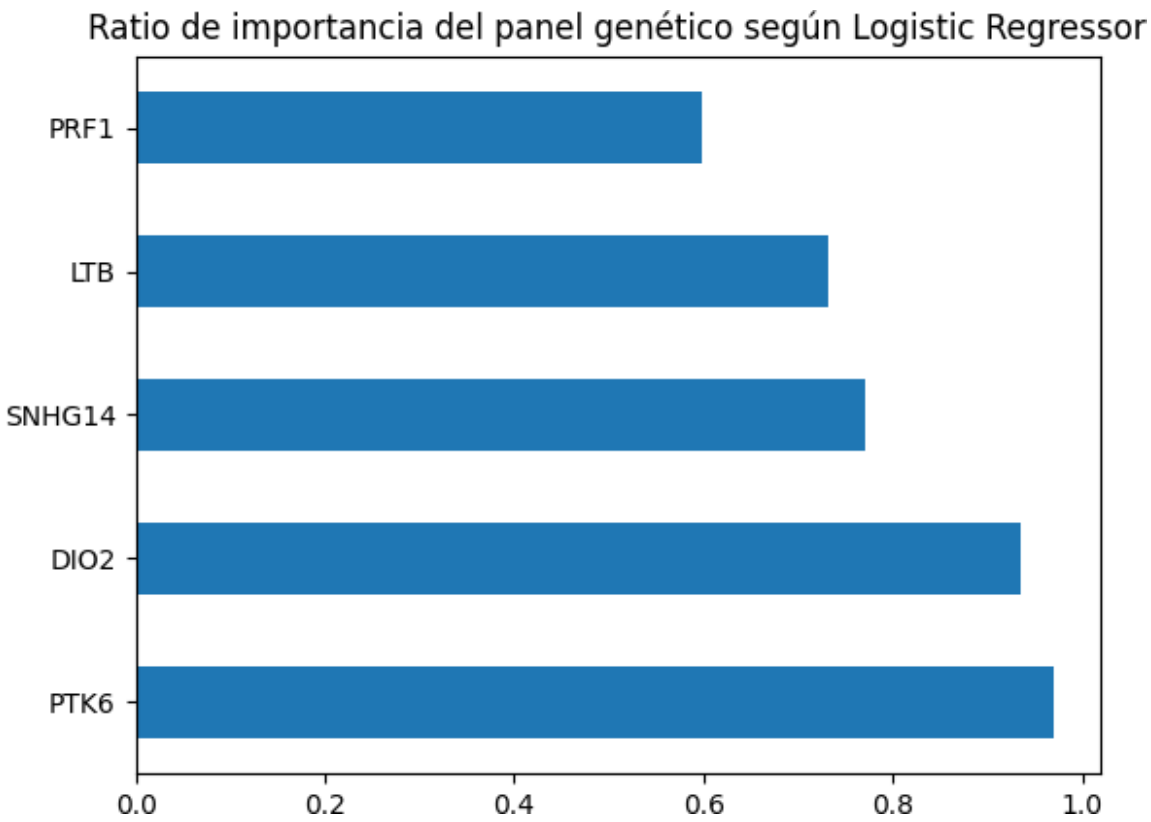
- `penalty` un valor a elegir entre `{'l1', 'l2', 'elasticnet', 'none'}`. Se usa para especificar la norma usada en la función de penalización para ajustar los coeficientes (por defecto = `'l2'`)
- `C` un valor real (`float`) para forzar mayor o menor sobreaprendizaje, valores altos implican un mejor ajuste en entrenamiento (por defecto = `1.0`)
- `class_weight` a elegir entre `{'balanced', None}` para dar o no mayor importancia a las muestras de clases menos representadas (por defecto = `None`)

Aunque no se indique explícitamente, muchos de los clasificadores implementados en *Scikit-Learn* poseen por defecto el parámetro `class_weight` con el que se puede abordar el problema del desequilibrio de clases (*imbalanced classification*).

En el siguiente bloque de código, se repiten los mismos pasos realizados para `Linear Regression` pero con el clasificador `Logistic Regressor`. Obsérvese que el mismo grupo de variables (genes) se destacan como principales en ambos casos. Finalmente, el acierto obtenido por este segundo modelo es más alto, indicando la preferencia por este tipo de soluciones.

```
import warnings
warnings.filterwarnings("ignore") #ignorar esta linea

lrm = linear_model.LogisticRegression()

lrm.fit(X_train, y_train.to_numpy().ravel())
print("Coef. independiente:",lrm.intercept_)
print("Coef. por variable:",lrm.coef_)
print()

importancia = np.abs(lrm.coef_[0]) #transformamos a una lista uni-
dimensional
importancia_norm = normalize(importancia[:,np.newaxis],
axis=0).ravel()
#Se representan las 5 más importantes según valor absoluto
(pd.Series(importancia,
index=X_train.columns).nlargest(5).plot(kind='barh'))
plt.title("Ratio de importancia del panel genético según Logistic
Regressor")
plt.show()
```

```
print()

y_pred = lrm.predict(X_test)

print("Valores de predicción originales (10 primeros):")
print(y_pred[:10])
print()

acc_score = accuracy_score(y_test, y_pred)
print("Acierto de Logistic Regression en la partición de test:",
acc_score)

Coef. independiente: [2.60557275]
Coef. por variable: [[-0.03110064  0.16743501 -0.58385565 -0.041624
0.40949269 -0.3470138
  -0.25777581 -0.28951875 -0.41787594 -0.31815611 -0.42031786 -
0.21280413
   0.12050644 -0.59472177 -0.05220371  0.14025638 -0.54429615 -
0.0470958
   0.30750881 -0.07255672  0.28527669  0.24941566 -0.3904066
0.19336178
   0.12320901  0.10120174 -0.14906486  0.37664058 -0.02005865
0.37683877
   0.33847116 -0.16205854  0.73224376  0.15976062 -0.2941222
0.07575672
  -0.24837593  0.26439539  0.54176416  0.77176051 -0.20439398 -
0.34556627
   0.5973885   0.96986976  0.09615201 -0.21446275 -0.93445767
0.03523568
  -0.31621746  0.25407509]]
```



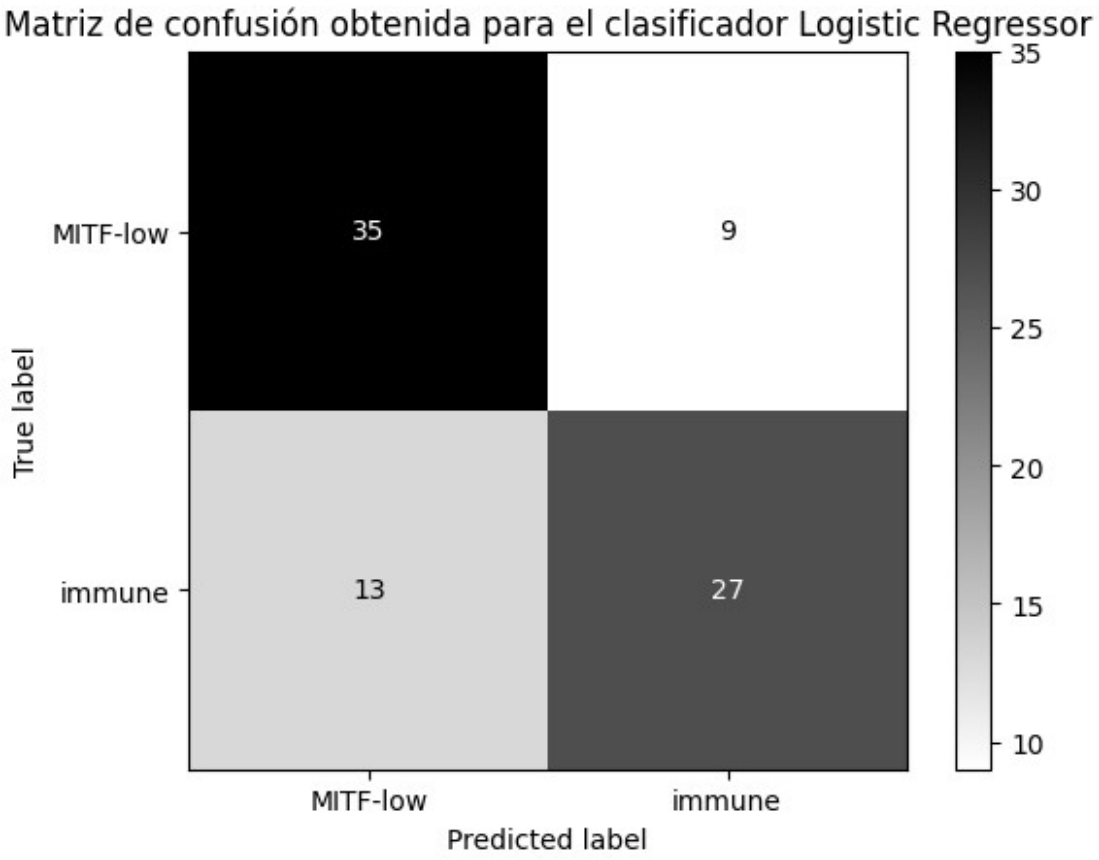

```
Valores de predicción originales (10 primeros):
['MITF-low' 'immune' 'MITF-low' 'MITF-low' 'immune' 'MITF-low' 'MITF-
low'
 'MITF-low' 'immune' 'immune']

Acierto de Logistic Regression en la partición de test:
0.7380952380952381
```

Además de la medida clásica de *accuracy*, se pueden utilizar otras diferentes como la medida *F1* que compensa entre los aciertos de cada clase, o la medida AUC. Utilizando un estimador o clasificador de *Scikit-Learn* es inmediato realizar la representación gráfica.

En el siguiente trozo de código, se repasan varios modos de obtener las métricas antes mencionadas:

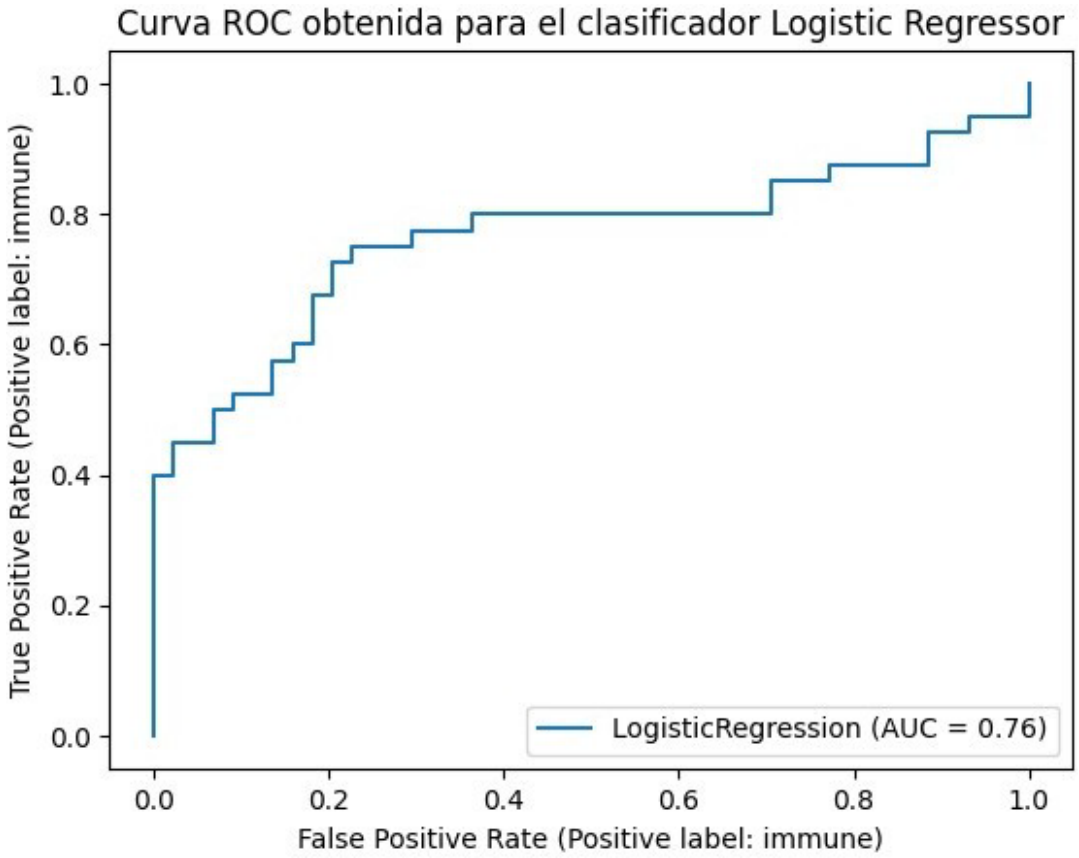
```
import matplotlib.pyplot as plt
from sklearn import metrics

metrics.ConfusionMatrixDisplay.from_estimator(lrm, X_test,
y_test,cmap='binary')
plt.title("Matriz de confusión obtenida para el clasificador Logistic
Regressor")
plt.show()
```

```
print(metrics.classification_report(y_test,y_pred))

f1 = metrics.f1_score(y_test,y_pred,pos_label="immune")
print("La medida F1 para el clasificador %s es %.4f"%
(lrm.__class__.__name__,f1))

y_probs = lrm.predict_proba(X_test)
auc = metrics.roc_auc_score(y_test, y_probs[:,1])
print("La medida AUC para el clasificador %s es %.4f"%
(lrm.__class__.__name__,auc))
metrics.RocCurveDisplay.from_estimator(lrm, X_test, y_test)
plt.title("Curva ROC obtenida para el clasificador Logistic
Regressor")
plt.show()
```




```
              precision    recall  f1-score   support

    MITF-low       0.73      0.80      0.76        44
      immune       0.75      0.68      0.71        40

    accuracy                           0.74        84
   macro avg       0.74      0.74      0.74        84
```

```
weighted avg        0.74      0.74      0.74         84

La medida F1 para el clasificador LogisticRegression es 0.7105
La medida AUC para el clasificador LogisticRegression es 0.7642
```



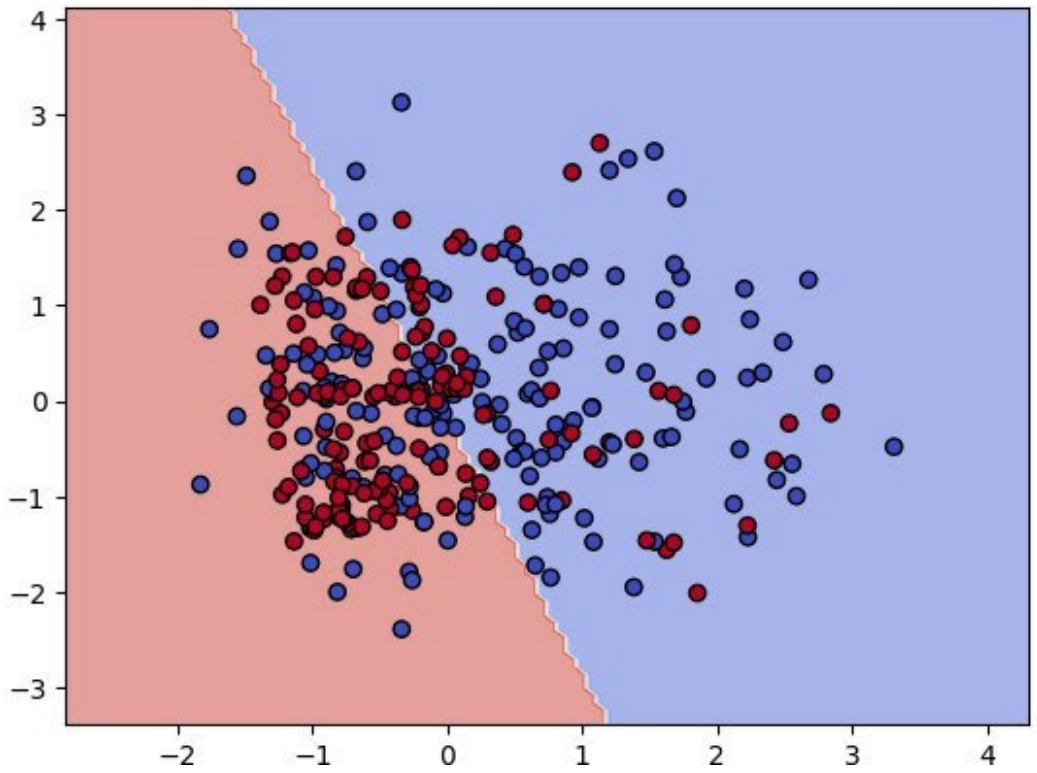
En lo sucesivo, se va a representar la frontera de decisión obtenida por cada clasificador empleado. Lo más importante con respecto a la representación de la función de decisión o discriminante, es que facilitan la comprensión sobre cómo funcionan realmente los distintos paradigmas de clasificación.

Es muy probable que no coincida exactamente el `accuracy` obtenido con respecto al conjunto de datos original, pero es entendible puesto que hemos transformado el problema.

```
from sklearn.inspection import DecisionBoundaryDisplay

clf = linear_model.LogisticRegression()
clf.fit(X_2D_train, y_2D_train)

fig = plt.figure(figsize=(12,9))
# Representamos la frontera de decisión
disp = DecisionBoundaryDisplay.from_estimator(clf, X_2D,

response_method="predict",
```

```
                                                    alpha=0.5,
                                                    cmap=plt.cm.coolwarm)

# Proyectamos encima los puntos del conjunto de datos
disp.ax_.scatter(X_2D[:, 0], X_2D[:, 1],
                 c=y_int, edgecolor="k",
                 cmap=plt.cm.coolwarm)
plt.show()
```

```
<Figure size 1200x900 with 0 Axes>
```



2.3 Ventajas e inconvenientes de los modelos lineales

Los modelos lineales poseen una serie de ventajas de gran interés para su uso práctico:

- Son muy eficientes en su uso y por tanto resultan muy apropiados para una aproximación inicial al problema.
- Una ventaja adicional es que son altamente interpretables, en el sentido que permiten determinar el peso o importancia de las variables asociadas a la clasificación. De este modo, se puede comprobar si los variables de entrada contienen un sentido biológico adecuado al caso de estudio.
- Por último, suelen funcionar bien incluso en problemas con un alto número de variables, ya que en ambos modelos presentados, las estimaciones de los coeficientes se basan en la independencia de las variables de entrada.

Debido al anterior motivo, hay que tener presente que en caso que estas variables de entrada estén correlacionados, la estimación resulta altamente sensible a los errores aleatorios en la variable de salida, produciendo una gran varianza. Por este motivo, si los datos del problema son relativamente complejos, es recomendable utilizar técnicas no lineales más sofisticadas.

3. CLASIFICACIÓN CON EL VECINO MÁS CERCANO (KNN)

En la cápsula anterior de este módulo, se introdujo brevemente este algoritmo de clasificación debido a su sencillez. A pesar de ello, su aplicación resulta generalizada en muchos problemas de Ciencia de Datos y Machine Learning.

En esta sección, se describe con más detalle el funcionamiento del algoritmo de vecino más cercano, para posteriormente conocer su uso mediante la implementación de Python en *Scikit-Learn*. Por último, se enumeran una serie de ventajas y desventajas relacionadas con esta técnica de clasificación.

3.1 Funcionamiento del algoritmo de vecino más cercano.

El algoritmo del vecino más cercano se encuadra en lo que se conoce como *Lazy Learning* o "aprendizaje perezoso" y es que realmente no existe fase de entrenamiento como tal. En lugar de ello, se realiza un "aprendizaje basado en instancias", es decir, el modelo es simplemente un almacén para las instancia de los datos de entrenamiento.

La premisa para realizar la clasificación sobre una nueva instancia se basa en analizar la clase para instancia *similares*. En términos más sencillos, se diría que, si "camina como un pato, y grazna como un pato, seguramente sea un pato".

La clave por tanto reside en qué se entiende por similitud entre instancias. Para ello, debemos definir lo que se conoce como función de distancia, que asignará un valor de salida entre instancias de acuerdo a cuán parecidos sean. Debido a que las instancias se representan en forma de variables numéricas, la más común de estas funciones de distancia será la distancia euclídea, que se describe mediante la siguiente ecuación:

\begin{equation} d_e(e_1, e_2) = \sqrt{\sum_{i=1}^{n}{(e_1^i - e_2^i)^2}} \end{equation}

Obsérvese que en el caso de dos dimensiones, coincidiría con la fórmula para hallar la longitud de la hipotenusa en un triángulo rectángulo, siendo los vértices de los catetos los puntos sobre los que desea encontrar su distancia.

En el algoritmo de vecino más cercano, para determinar la clase de salida de una nueva instancia, se debe calcular el valor de distancia para todas las instancias disponibles en el conjunto entrenamiento. A continuación, se asigna la clase más frecuente de entre los k vecinos más cercanos a la nueva instancia.

K es un parámetro crítico en el uso de este algoritmo de clasificación. Siempre debe ser un valor impar para evitar posibles empates. No obstante, la elección del valor óptimo de k depende en gran medida de los datos: en general, un valor mayor suprime los efectos del posible ruido, pero hace que los límites de la clasificación sean menos claros.

3.2 Implementación en Scikit-Learn y principales parámetros de uso

Recuérdese que la implementación del vecino más cercano en *Scikit-learn* se encuentra en el método `KNeighborsClassifier` (del inglés *Clasificador de K Vecinos*). Los parámetros más importantes que se pueden configurar en este método son los siguientes:

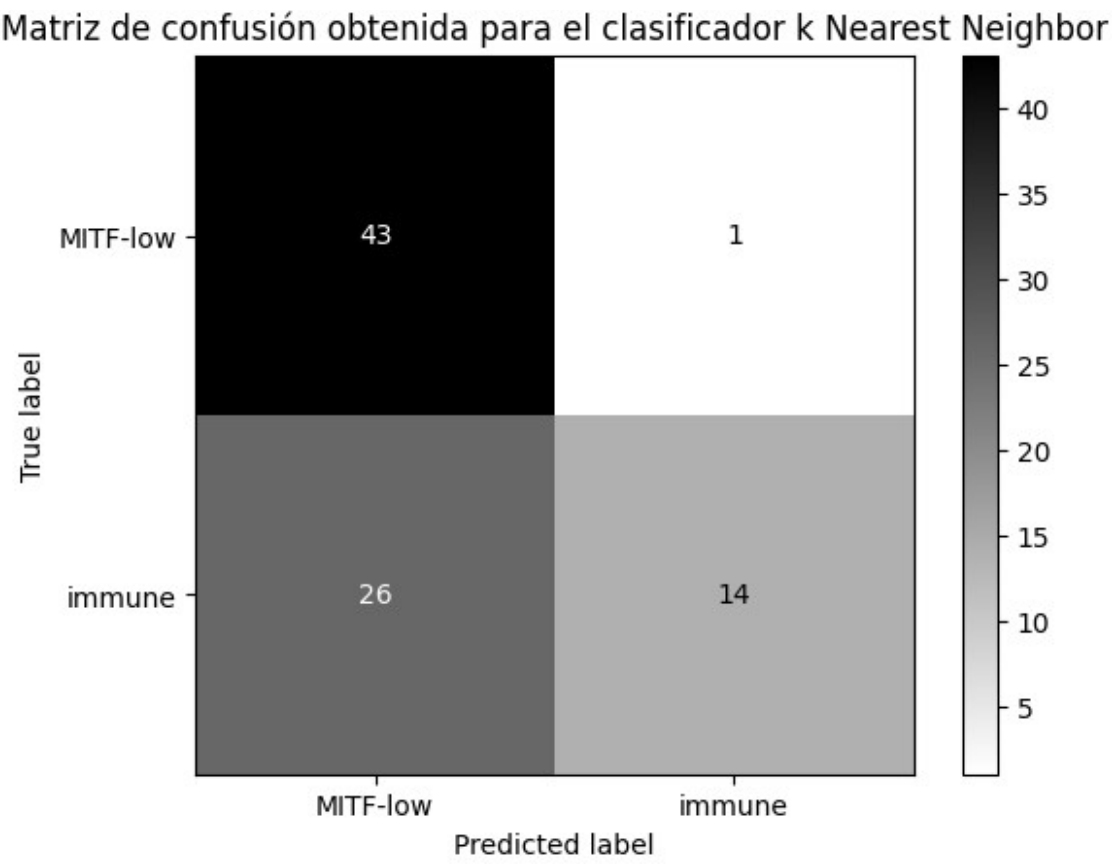
- `n_neighbors` un valor entero (`int`) para determinar el entorno del vecindario (por defecto = `5`)
- `weights` a elegir entre `{'uniform', 'distance'}` según se quiera que la salida sea por voto simple, o la etiqueta de salida de los vecinos más cercanos tenga mayor importancia, respectivamente (por defecto = `'uniform'`)
- `metric`: a elegir entre {'euclidean', 'manhattan', 'chebyshev', 'minkowski', 'mahalanobis'}, para determinar el cálculo de la distancia (por defecto `'minkowski'`)

En el siguiente ejemplo se observa su funcionamiento, tomando *k* (parámetro `n_neighbors`) igual a 3, y el resto por defecto.

```
from sklearn.neighbors import KNeighborsClassifier # cargamos la
función desde la biblioteca

#Ejemplo de uso de kNN
knn = KNeighborsClassifier(n_neighbors=3, metric='euclidean') #
instanciamos el modelo
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)

acc_score = accuracy_score(y_test, y_pred)
print("Acierto de KNN en la partición de test:", acc_score)

Acierto de KNN en la partición de test: 0.6785714285714286
```

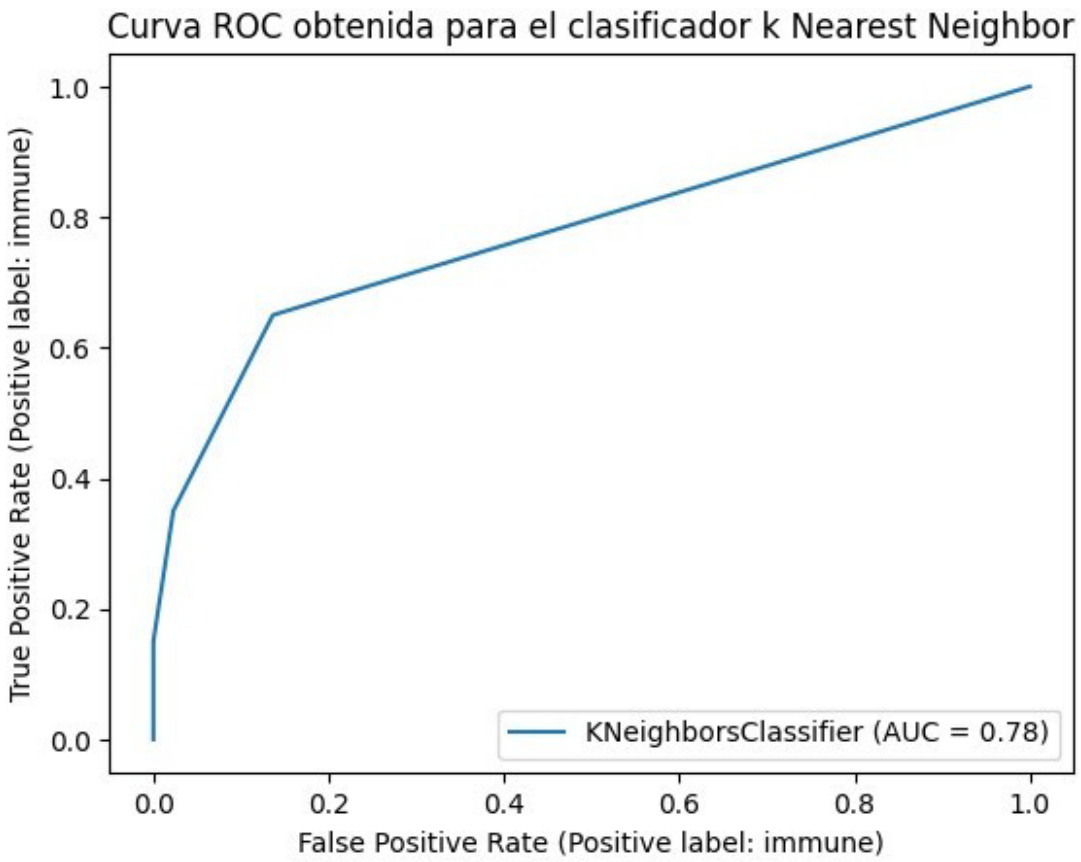
Se repite el modo de obtener otras métricas de calidad:

```
metrics.ConfusionMatrixDisplay.from_estimator(knn, X_test,
y_test,cmap='binary')
plt.title("Matriz de confusión obtenida para el clasificador k Nearest
Neighbor")
plt.show()

print(metrics.classification_report(y_test,y_pred))

f1 = metrics.f1_score(y_test,y_pred,pos_label="immune")
print("La medida F1 para el clasificador %s es %.4f"%
(knn.__class__.__name__,f1))

y_probs = knn.predict_proba(X_test)
auc = metrics.roc_auc_score(y_test, y_probs[:,1])
print("La medida AUC para el clasificador %s es %.4f"%
(knn.__class__.__name__,auc))
metrics.RocCurveDisplay.from_estimator(knn, X_test, y_test)
plt.title("Curva ROC obtenida para el clasificador k Nearest
Neighbor")
plt.show()
```



	precision	recall	f1-score	support
MITF-low	0.62	0.98	0.76	44
immune	0.93	0.35	0.51	40
accuracy			0.68	84
macro avg	0.78	0.66	0.64	84
weighted avg	0.77	0.68	0.64	84

```
La medida F1 para el clasificador KNeighborsClassifier es 0.5091
La medida AUC para el clasificador KNeighborsClassifier es 0.7750
```



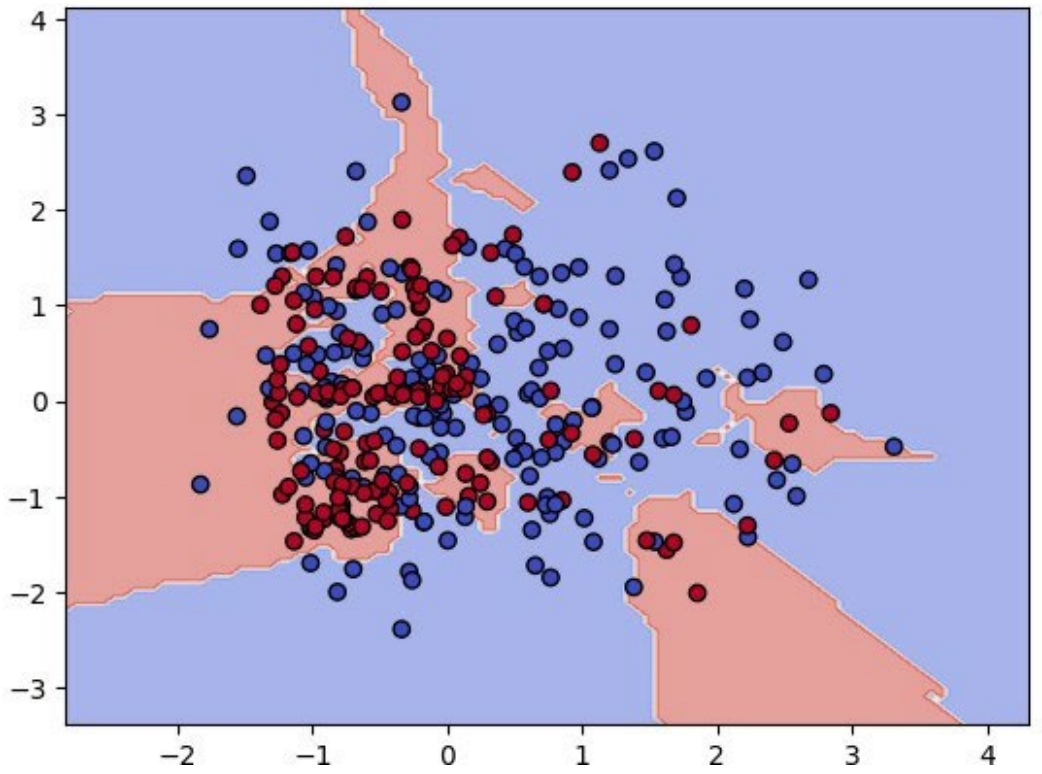
Nuevamente, podemos comprobar cómo se definen las fronteras de clasificación para este algoritmo en particular. Nótese el tipo de función discriminante no lineal (deja de ser una "recta"), y en muchas ocasiones además local (muy "cerrado") en áreas concretas del espacio de datos. A diferencia de los modelos lineales anteriores, kNN basa su funcionamiento en el entorno de los ejemplos.

El tiempo cálculo para obtener la representación gráfica podría resultar bastante mayor en este caso particular, dadas las características de eficiencia del algoritmo.

```
#Creamos y entrenamos el clasificador con los datos 2D
clf = KNeighborsClassifier()
clf.fit(X_2D_train, y_2D_train)
score = clf.score(X_2D_test,y_2D_test)

fig = plt.figure(figsize=(12,9))
# Representamos la frontera de decisión
disp = DecisionBoundaryDisplay.from_estimator(clf, X_2D,

response_method="predict",
                                              alpha=0.5,
                                              cmap=plt.cm.coolwarm)

# Proyectamos encima los puntos del conjunto de datos
```

```
disp.ax_.scatter(X_2D[:, 0], X_2D[:, 1],
                 c=y_int, edgecolor="k",
                 cmap=plt.cm.coolwarm)
plt.show()
```

```
<Figure size 1200x900 with 0 Axes>
```



3.3 Ventajas e inconvenientes del algoritmo kNN

Por último, se analizan las ventajas y desventajas de este algoritmo de clasificación, que deberán ser tenidas en cuenta de acuerdo al caso de estudio que se esté analizando.

- Ventajas:
  - Funciona bien incluso con instancias ruidosas (cuyo valor puede ser erróneo), utilizando como parámetro un valor de `k` moderado (`k`>1). Esto implica que en caso que haya datos anómalos (instancias que puedan contener valores erróneos), éstos no deberían afectar a la salida del clasificador, puesto que se compensaría con el resto de datos que sí son correctos.
  - Bastante eficaz: utiliza funciones locales lineales para aproximarse a la función objetivo. La frontera es no lineal y depende del muestreo de datos que se haya realizado.
  - Válido tanto para la clasificación como para la regresión.
  - Se puede utilizar fácilmente con prototipos. Esto significa que, si en lugar de utilizar todo el conjunto de entrenamiento, se pudiese hacer una selección de las

instancias más importantes, el algoritmo sería mucho más rápido, sin perder capacidad predictiva.
  - Disponible en la mayoría de los paquetes de software.
- Desventajas:
  - Muy ineficiente en la memoria, ya que todo el conjunto de datos debe ser almacenado en el sistema.
  - La complejidad computacional es $O(d \cdot n^2)$ con $O(d)$ la complejidad de la distancia. Por tanto, a mayor número de instancias, más lenta resulta la predicción.
  - La distancia entre vecinos puede estar dominada por variables de entrada irrelevantes. Si no se realiza una correcta selección de variables, puede afectar en un alto grado a la predicción.
  - La distancia converge al mismo valor en una alta dimensionalidad. Esto significa que cuando se tiene un alto número de variables, la distancia o similaridad tiende a ser idéntica para cualquier instancia sobre la que se calcule.

4. ÁRBOLES DE DECISIÓN

Los árboles de decisión (en inglés *Decision Trees* o DT) son una técnica de aprendizaje supervisado utilizado tanto para clasificación y la regresión.

En esta sección, primero se presentan las propiedades de las técnicas basadas en árboles de decisión. A continuación, se describen los pasos necesarios para utilizar la implementación disponible en la biblioteca Scikit-Learn. Por último, se enumeran una serie de ventajas e inconvenientes asociadas a los árboles de decisión en general.

4.1 Introducción a árboles de decisión en clasificación

Un modelo de este tipo se basa en reglas de decisión simples, o condiciones, con el formato "*si-entonces-sino*" (`IF-THEN-ELSE`), normalmente dicotómicas (grupos de dos). Existe un orden jerárquico en la aplicación de las reglas, que se van encadenando hasta dar la decisión final. Por este motivo, la forma habitual de estructurar el modelo es en forma de árbol, de ahí su nombre. Sus elementos son los siguientes:

- Cada hoja es una categoría (clase) correspondientes a la salida.
- Cada nodo (parte interna del árbol) especifica una prueba simple a realizar (regla sobre una única tupla <variable, valor>).
- Los descendientes de cada nodo son los posibles resultados de la prueba del nodo

Durante el aprendizaje, se busca que cada nodo realice una división disjunta sobre el número de instancias de cada clase. Este criterio de división se calcula mediante dos posibles medidas: la ganancia de información (*entropía*) y la impureza (*índice Gini*). En definitiva, se busca la combinación <variable, valor> que minimiza o maximiza los valores anteriores (entropía o gini, respectivamente), lo que implica que todas las instancias de una clase se "agrupan" sobre la parte "SI" de la regla, mientras que el resto irán a la parte "SINO".

De acuerdo a lo anterior, las variables de entrada más importantes aparecerán en la parte superior (raíz) del árbol (en el ejemplo de la imagen anterior sería el color). Esto es debido a que son las que se escogen inicialmente como más adecuadas para separar entre las clases del problema.

El proceso de entrenamiento normalmente se realiza de manera recursiva: se comienza identificando la raíz, y se continua hasta llegar a cada una de las hojas del árbol. Se determina que se ha llegado a una hoja (y por tanto no proceden más divisiones) de acuerdo a dos posibles razones:

- Se ha alcanzado un umbral de pureza del nodo, marcado por el valor mínimo o máximo de la medida de división (entropía o ganancia).
- El árbol ha llegado a un límite de profundidad máxima marcada por el usuario. La profundidad se mide con el número de nodos desde la raíz hasta la hoja.

Normalmente, cuanto más profundo es el árbol, más complejas son las reglas de decisión y más ajustado es el modelo. No obstante, se debe tener cuidado ya que este ajuste puede llevar al sobreaprendizaje.

4.2 Implementación en Scikit-Learn y principales parámetros de uso

La clase de Scikit-Learn que incluye árboles de decisión se llama `DecisionTreeClassifier`. Su uso es idéntico al resto de clasificadores implementados en esta biblioteca, por lo que nos centraremos en sus principales parámetros adicionales:

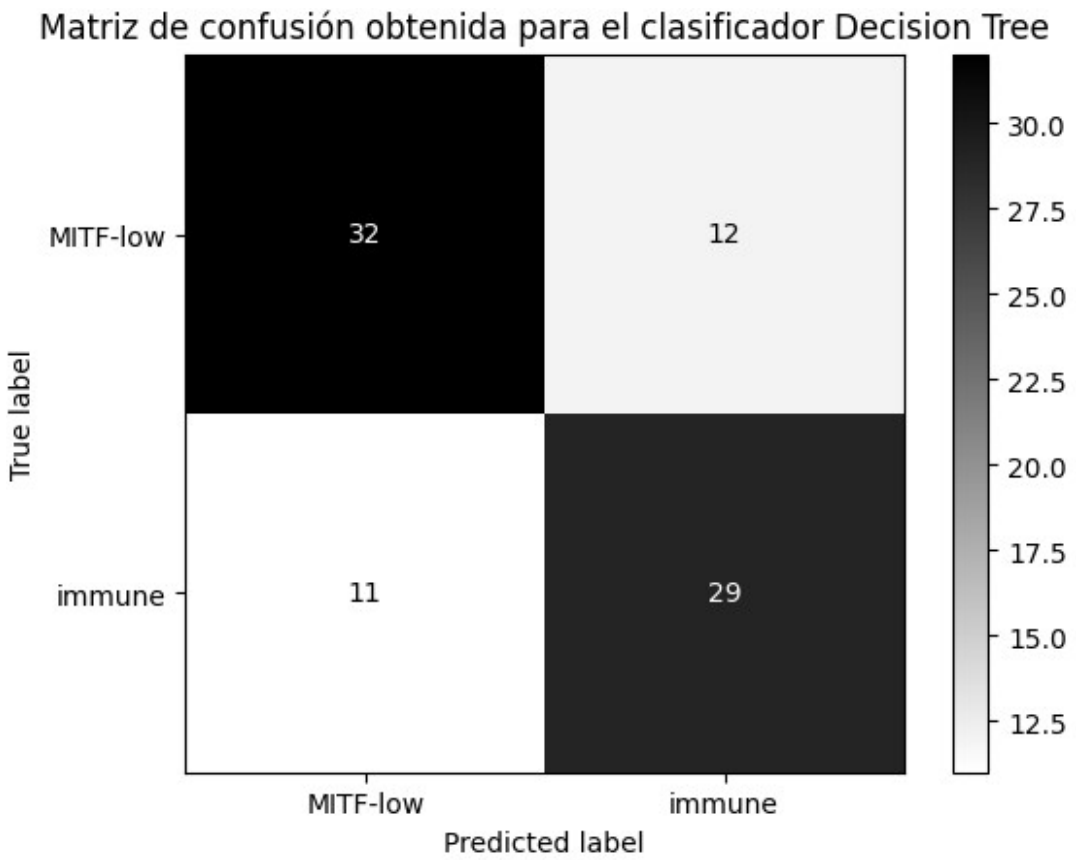
- `criterion` a elegir entre `{"gini", "entropy"}`, para determinar la función que mide la calidad de una división (por defecto=`"gini"`).

- `max_depth` es un número entero (`int`) que determinar la máxima profundidad del árbol. Si no se indica, entonces los nodos se expanden hasta que todas las hojas sean puras o hasta que contengan menos de `min_samples_split` instancias.
- `min_samples_split` es un número entero (`int`) que indica el número mínimo de instancias necesarias para dividir un nodo interno (por defecto=`2`)

```
from sklearn import tree

dt = tree.DecisionTreeClassifier()
dt.fit(X_train, y_train)

y_pred = dt.predict(X_test)

acc_score = accuracy_score(y_test, y_pred)
print("Acierto de DT en la partición de test:", acc_score)
```

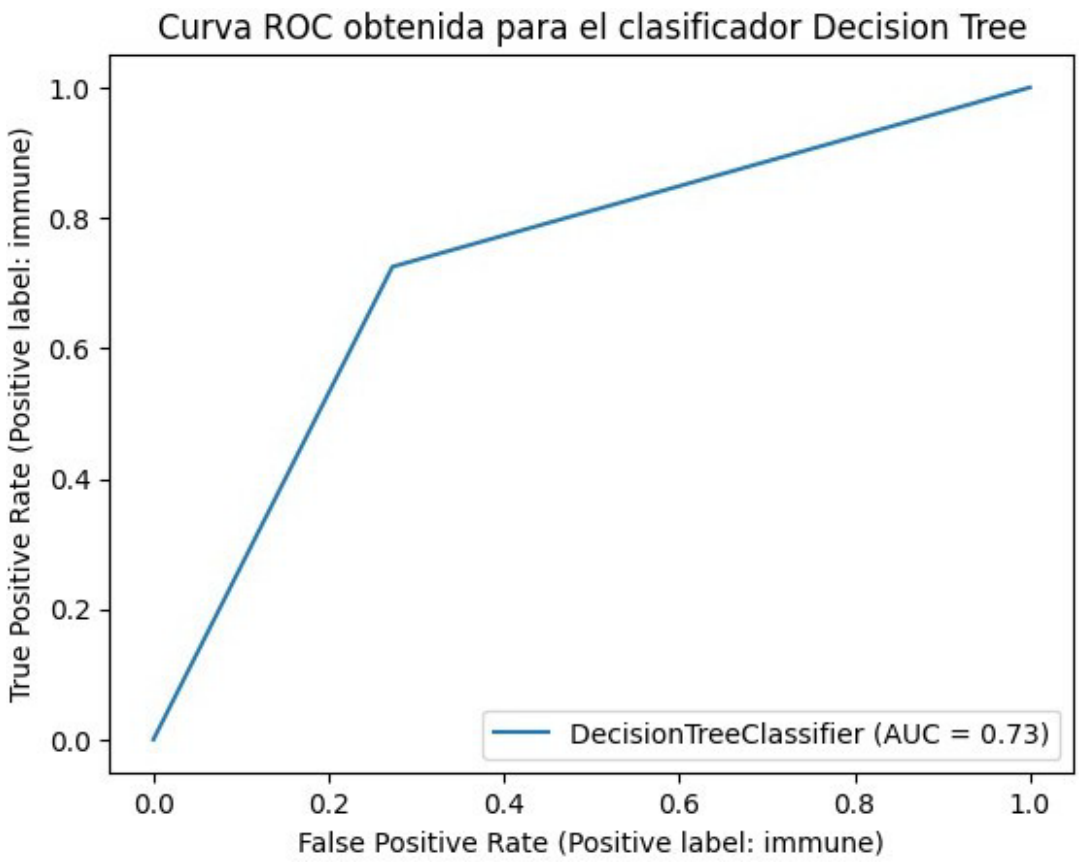
```
Acierto de DT en la partición de test: 0.7261904761904762
```

```
metrics.ConfusionMatrixDisplay.from_estimator(dt, X_test,
y_test,cmap='binary')
plt.title("Matriz de confusión obtenida para el clasificador Decision
Tree")
plt.show()

print(metrics.classification_report(y_test,y_pred))

f1 = metrics.f1_score(y_test,y_pred,pos_label='immune')
print("La medida F1 para el clasificador %s es %.4f"%
(dt.__class__.__name__,f1))

y_probs = dt.predict_proba(X_test)
auc = metrics.roc_auc_score(y_test, y_probs[:,1])
print("La medida AUC para el clasificador %s es %.4f"%
(dt.__class__.__name__,auc))
metrics.RocCurveDisplay.from_estimator(dt, X_test, y_test)
plt.title("Curva ROC obtenida para el clasificador Decision Tree")
plt.show()
```



	precision	recall	f1-score	support
MITF-low	0.74	0.73	0.74	44
immune	0.71	0.72	0.72	40
accuracy			0.73	84
macro avg	0.73	0.73	0.73	84
weighted avg	0.73	0.73	0.73	84

```
La medida F1 para el clasificador DecisionTreeClassifier es 0.7160
La medida AUC para el clasificador DecisionTreeClassifier es 0.7261
```



Una de las grandes ventajas de los árboles de decisión es su buena interpretabilidad. Debido al uso de reglas sencillas en formato de árbol, es muy fácil determinar sus principales componentes, conocer las variables más útiles que representan el caso de estudio, o incluso dar una explicación directa para cada salida realizada por el clasificador, dado que será un único camino de la raíz al nodo hoja.

En el siguiente bloque, se realiza una visualización del árbol generado, donde además se puede indicar explícitamente el nombre de las variables de entrada, así como las clases. Los colores (azul y naranja) indican las clases de salida mayoritarias en cada nodo, donde a mayor grado de color indica mejor separación de clases.

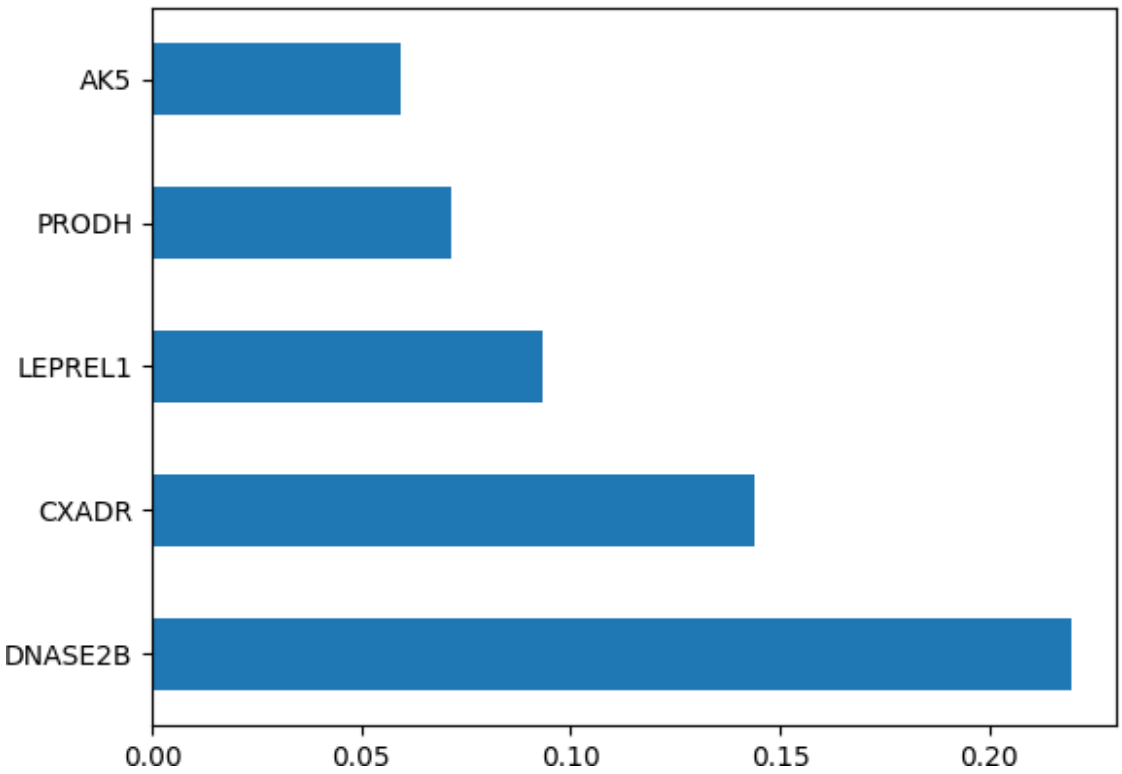
```
#Bibliotecas necesarias para una mejor visualización
from graphviz import Source

#se pinta el árbol:
tree_graph = tree.export_graphviz(dt, out_file=None,
                                  feature_names=X.columns,

class_names=pd.unique(y[y.columns[0]]),
                                  filled = True)
graph = Source(tree_graph)
graph
```

De igual modo que se realizó con el clasificador `LogisticRegressor` con un árbol de decisión resulta sencillo comprobar cuáles son las variables más importantes en el modelo. Por normal general, son aquéllas más cercanas al nodo raíz, pero existe también una propiedad de la clase `DecisionTreeClassifier` que nos permite acceder a esta información.

Se puede observar que la variable seleccionada como la más importante de todas es justamente la que aparece en la raíz del árbol. Esto no es casualidad, debido a que es la que primero se selecciona para dividir las clases.

Por último, puesto que el aprendizaje del modelo es diferente al del paradigma de clasificación lineal, el ranking de las variables más importantes también cambia.

```
#En primer lugar, se capturan los valores del ranking de importancia
importancia = dt.feature_importances_
#Se representan las 5 más importantes
(pd.Series(importancia,
index=X_train.columns).nlargest(5).plot(kind='barh'))
plt.show()
```

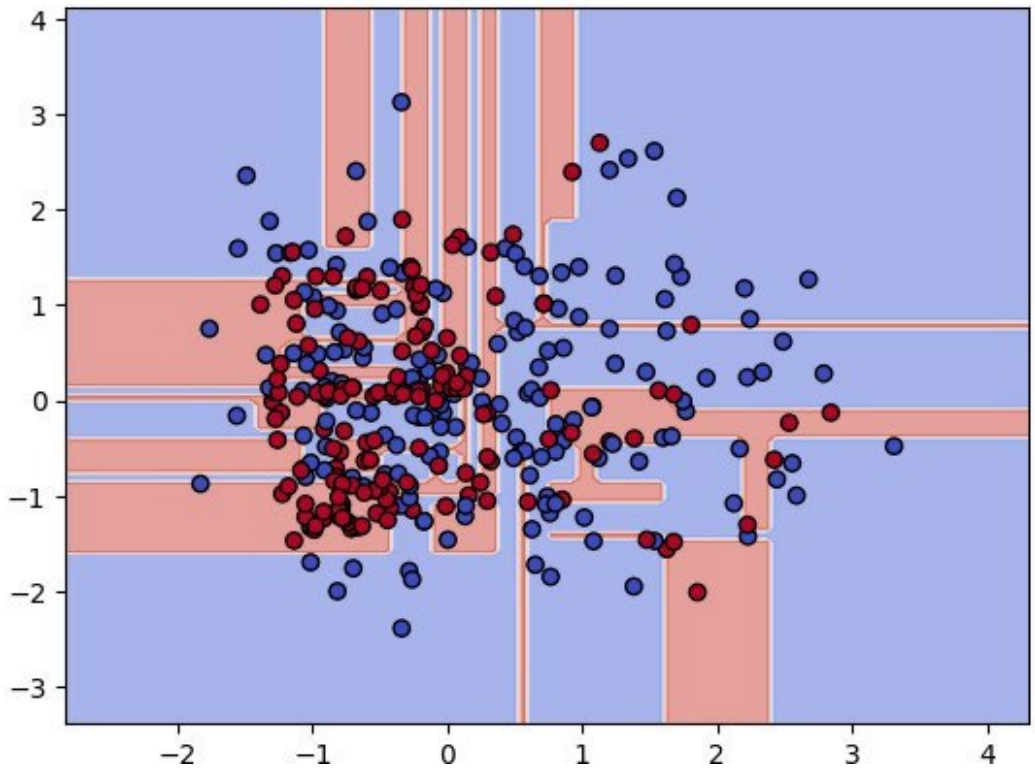


Por último, se procede a representar el tipo de frontera de decisión obtenida por el paradigma de clasificación basado en árboles de decisión.

En esta ocasión, se aprecia una división por bloques rectangulares, de acuerdo a cómo se distribuyen los ejemplos en cada nodo del árbol.

```
#Creamos y entrenamos el clasificador con los datos 2D
clf = tree.DecisionTreeClassifier()
clf.fit(X_2D_train, y_2D_train)
score = clf.score(X_2D_test,y_2D_test)

fig = plt.figure(figsize=(12,9))
# Representamos la frontera de decisión
disp = DecisionBoundaryDisplay.from_estimator(clf, X_2D,

response_method="predict",
                                              alpha=0.5,
                                              cmap=plt.cm.coolwarm)

# Proyectamos encima los puntos del conjunto de datos
disp.ax_.scatter(X_2D[:, 0], X_2D[:, 1],
                 c=y_int, edgecolor="k",
                 cmap=plt.cm.coolwarm)
plt.show()
```

```
<Figure size 1200x900 with 0 Axes>
```



4.3 Ventajas e inconvenientes de los árboles de decisión

Los árboles de decisión son una de las herramientas más potentes utilizadas en Machine Learning. Entre sus principales ventajas, se pueden destacar las siguientes:

- Fácil de usar y eficiente. No contiene un excesivo número de parámetros, y éstos son sencillos de entender y configurar, para adaptarse mejor al problema. Además, es muy rápido en su ejecución, por lo que permite realizar continuas pruebas.
- Las reglas generadas son fáciles de interpretar. Se ha comentado que una de las principales virtudes de los árboles de decisión es que son sistemas comprensibles por el usuario, puesto que utilizan reglas de cognición similares a las que aplicaría el experto.
- Escalan mejor que otro tipo de técnicas. Si aumentamos el número de instancias o de variables, el rendimiento en tiempo de cómputo no se verá excesivamente afectado.
- Puede manejar posibles datos ruidosos. Para ello, utiliza un mecanismo interno conocido como "*poda*" que permite reducir la profundidad del árbol de manera heurística en aras de una mejor generalización.

El número de aspectos positivos de los árboles de decisión es bastante amplio; sin embargo, se deben tener en cuenta algunos detalles que pueden afectar a su uso:

- No maneja directamente las variables de entrada de tipo numérico. Para el cálculo de las funciones de entropía o gini, las variables se deben discretizar previamente. Esto es transparente al usuario, pero debe ser tenido en cuenta.
- Intenta dividir el dominio de la variable en regiones rectangulares. Este tipo de frontera de decisión puede no ser adecuada en algunas distribuciones de salida que sean lineales.

- Tienen dificultades para lidiar con los valores perdidos. Se hace necesario imputar previamente estos valores.
- Puede tener problemas de sobreaprendizaje. Si se aplica un factor muy alto de profundidad, será más dificil que generalice correctamente sobre las instancias de test.
- No se detectan correlaciones entre las variables. Cada nodo de decisión se obtiene de forma independiente, sin tener en cuenta al resto.

REFERENCIAS BIBLIOGRÁFICAS

- Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques. San Francisco, CA, USA: Morgan Kaufmann Publishers. ISBN: 0123814790, 9780123814791
- Witten, I. H., Frank, E., Hall, M. A., Pal, C. J. (2017). Data mining: practical machine learning tools and techniques. Amsterdam; London: Morgan Kaufmann. ISBN: 9780128042915 0128042915
- Scikit-Learn: Supervised Learning https://scikit-learn.org/stable/supervised_learning.html (visitado el 25 de Junio de 2020).
- Open Machine Learning Course: Topic 3. Classification, Decision Trees and k Nearest Neighbors https://mlcourse.ai/articles/topic3-dt-knn/ (visitado el 25 de Junio de 2020).

Referencias adicionales

- Alpaydin, E. (2016). Machine Learning: The New AI. MIT Press. ISBN: 9780262529518
- Towards Data Science: The Complete Guide to Classification in Python https://towardsdatascience.com/the-complete-guide-to-classification-in-python-b0e34c92e455 (visitado el 25 de Junio de 2020).
- Towards Data Science: Python For Data Science — A Guide To Classification Machine Learning https://towardsdatascience.com/python-for-data-science-a-guide-to-classification-machine-learning-9ff51d237842 (visitado el 25 de Junio de 2020).