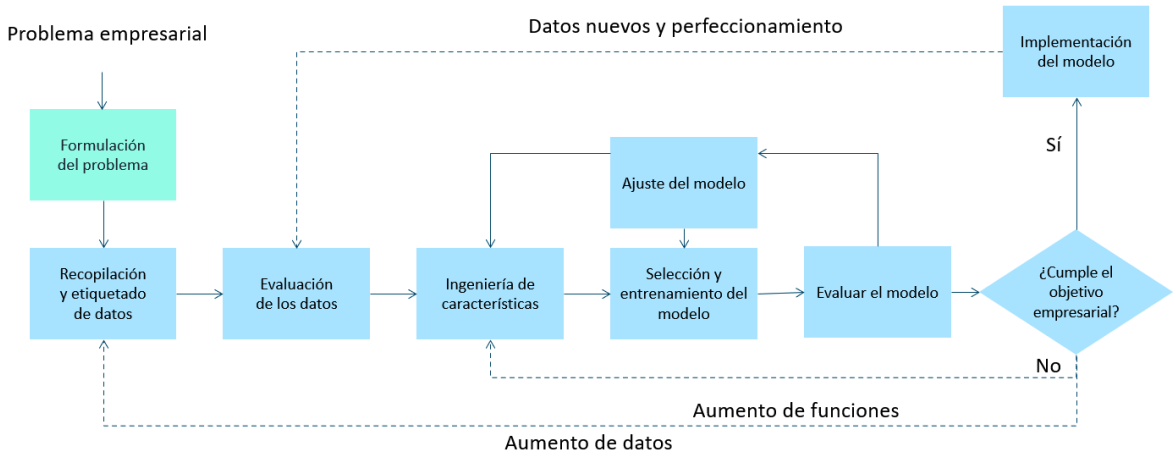
Unidad 3. El proceso de trabajo en Machine Learning



Módulo: “Programación de Inteligencia Artificial”

Curso de Especialización en Inteligencia Artificial y Big Data

IES Serra Perenxisa

Contenidos

0. Introducción

En esta unidad vamos a trabajar el ciclo completo en los 2 subtipos de aprendizaje supervisado más importante, dentro del Machine Learning (en adelante ML), como son la regresión y la clasificación.

Cuando necesitamos desarrollar una aplicación o modelo de inteligencia artificial en el mundo real normalmente los datos que necesitamos para nuestro modelo no están directamente disponibles en un formato adecuado. A menudo, incluso, estos datos provienen de diferentes fuentes de datos. En cualquier caso, debemos disponer de algún mecanismo para recuperar esos datos, limpiar las partes que no nos interesen y dejar el resto estructurado en un formato adecuado para nuestro programa. Además, conviene conocer la composición de estos datos y sus características relevantes. Este proceso general se suele componer al menos de tres etapas, que no necesariamente se deben realizar en el orden que aquí se indica.

1. **Limpieza de datos** (*data cleaning*): consiste en procesar los datos para eliminar las partes que no interesen y dar un formato adecuado a las que sí. Por ejemplo, gestionar los valores nulos que pueda haber, o los valores anómalos, ver el tipo de dato adecuado para cada columna, etc.
2. **Ingeniería de datos** (*data engineering*): consiste en generar nuevos datos a partir de los que ya existen. Aquí entran aspectos como la selección de características relevantes, la conversión de variables categóricas en numéricas, o el escalado de los datos para que todos tengan el mismo peso o importancia.
3. **Análisis exploratorio de datos** (*EDA*, *Exploratory Data Analysis*), que consiste en analizar los datos de que disponemos buscando patrones o información relevante: medias, modas, tendencias, distribución de valores, etc. Para ello se suele hacer uso de las representaciones gráficas que ya hemos trabajado con Matplotlib o Seaborn.

La calidad o precisión de las decisiones que pueda tomar un programa dependen en gran parte de la calidad de los datos que le proporcionamos de entrada. En general, podemos considerar que los datos de que disponemos son de calidad si se ajustan a las operaciones que sobre ellos se realizan. Dicho de otro modo, no hay un estándar para medir esa calidad de los datos y, en general, dependerá de su adecuación al programa en cuestión.

En este documento veremos algunos ejemplos de técnicas de tratamiento y análisis de datos que podemos aplicar en nuestros desarrollos, siempre como paso previo a la ingesta de datos por parte de la aplicación. Hay que tener en cuenta que, en muchas ocasiones, este proceso de tratamiento y limpieza de datos puede llevar mucho más tiempo que lo que supone el desarrollo del modelo en sí.

1. Presentación del caso de prueba

Tomaremos como base un dataset que contiene información sobre datos de salud de distintos pacientes: género, edad, peso, colesterol... Pretendemos desarrollar un modelo que pueda predecir la presión sanguínea alta de un paciente (también llamada *presión sistólica*, almacenada en la columna *ap_hi*) a partir del resto de información. Como veremos, el documento CSV tiene muchas características, pero algunas de ellas van a resultar irrelevantes para determinar el valor objetivo, y otras que sí van a ser relevantes tendrán datos incompletos que tendremos que gestionar.

Podemos crear un notebook de Jupyter y cargar el CSV para ir probando en él los pasos que explicaremos a continuación.

1.1. Carga de librerías

Comenzaremos importando las librerías necesarias para nuestro proyecto. Además de la librería `sklearn` incorporaremos algunos módulos que nos van a resultar de utilidad para ciertas tareas puntuales:

```
# Importar librerías
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
```

1.2. Carga del CSV y presentación inicial de los datos relevantes

A continuación, cargaremos los datos del archivo CSV, mostraremos su estructura en pantalla y haremos un primer análisis del dato objetivo (columna *ap_hi*, presión sanguínea alta). Deberás poner el CSV en tu carpeta de trabajo.

```
# Cargar datos de CSV en variable "datos"
datos = pd.read_csv('datos_salud2.csv')

# Guardarnos en una variable el nombre de columna objetivo "ap_hi"
valor_objetivo = 'ap_hi'

# Mostrar 5 primeras filas
print(datos.head())

# Mostrar tamaño del dataset (filas y columnas)
print(datos.shape)

# Mostrar información relevante de 'ap_hi' con su método 'describe'
print(datos[valor_objetivo].describe())
```

Como podemos observar tenemos 70.000 registros, donde el valor máximo es 16020, y el mínimo es -150, siendo el promedio 128.82 aproximadamente. Algunos de estos valores son muy anómalos, ya que una presión de 16020 es totalmente excesiva, y no se pueden tener presiones sanguíneas negativas. Más adelante veremos cómo gestionar estos valores.

2. Limpieza de datos (*data cleaning*)

Comenzaremos por el proceso de limpieza de datos, que consistirá fundamentalmente en detectar y corregir los valores nulos, las anomalías, y estudiar el tipo de dato más adecuado para cada columna. Estas correcciones son importantes debido al grave impacto que tiene para un modelo de IA la entrada de este tipo de errores.

2.1. Gestión de valores nulos (*missing values*)

En ocasiones es posible que algún campo o columna de nuestros datos tenga valores nulos o faltantes. Esto supone una pérdida de información, que se puede manifestar de distintas formas. Por ejemplo, si nos falta un valor numérico normalmente éste aparece como *NaN* (*Not a Number*) una sintaxis especial para identificar datos que se suponen numéricos pero no lo son. En otros casos podemos encontrarnos con un valor *NA* (*Not Available*) cuando ese dato en concreto no está disponible.

Por ejemplo, consideremos el siguiente listado de datos personales, donde falta la edad de una persona y el peso de otra:

Nombre	**Edad**	**Peso**
Juan	70	75
Ana	NaN	70
Mario	30	NaN
Laura	26	67

Dependiendo del problema podemos optar por descartar la fila con valores omitidos, o por intentar reemplazar el valor omitido por otro simulado que no sea discordante con el resto, como por ejemplo la media, mediana o moda.

2.1.1. Detección o conteo de valores nulos

La función `isnull` obtiene si una determinada casilla tiene o no valor. Podemos combinarla con la función `sum` para saber cuántas casillas vacías tenemos. Podemos construir un ejemplo como el anterior y ver cuántas casillas nulas hay:

```
import numpy as np
import pandas as pd

datosIniciales = {'Nombre': ['Juan', 'Ana', 'Mario', 'Laura'],
    'Edad': [70, np.NaN, 30, 26],
    'Peso': [75, 70, np.NaN, 67]}
```

```
datos = pd.DataFrame(datosIniciales)

print("Número de casillas nulas:")
print(datos.isnull().sum())
# Dirá que la columna "Nombre" no tiene valores perdidos
# y la columna "Edad" y "Peso" tienen un valor perdido cada una
```

Podríamos obtener, por ejemplo, todas las filas de un *data frame* que tengan nula una determinada casilla:

```
nulos_edad = datos[datos['Edad'].isnull()]
```

El siguiente fragmento de código construye una tabla donde, para cada columna, muestra el porcentaje de casillas nulas que tiene respecto al total de filas:

```
valores_nulos = datos.isnull().sum().sort_values(ascending=False)
# Añadimos "reset_index" para numerar las filas resultantes
ratio_nulos = (valores_nulos / len(datos)).reset_index()
ratio_nulos.columns = ['Caracteristica', 'RatioNulos']
ratio_nulos
```

2.1.2. Opciones ante la presencia de valores nulos

Si detectamos la presencia de valores nulos en una columna *X*, básicamente tenemos dos alternativas:

- Una consiste en **reemplazar** el valor nulo por un valor representativo, que puede ser por ejemplo la media o mediana de valores de la columna (en el caso de valores numéricos) o la moda (el valor que más se repite, útil para valores categóricos). También hay otras opciones, como reemplazar por el valor máximo o el mínimo de la columna. Es lo que se conoce como **imputación** de valores. Usaremos en cualquier caso el método `fillna` sobre la columna en cuestión, indicando el valor de reemplazo.

  ```
  # Reemplazo por ceros
  datos['X'].fillna(0, inplace=True)

  # Reemplazo por la media
  datos['X'].fillna(datos['X'].mean(), inplace=True)

  # Reemplazo por la mediana
  datos['X'].fillna(datos['X'].median(), inplace=True)

  # Reemplazo por la moda (devuelve un array con las ocurrencias mayores)
  datos['X'].fillna(datos['X'].mode()[0], inplace=True)
  ```

- Otra consiste en **eliminar** las filas o registros que contengan un valor nulo en esa columna. No es aconsejable en algunos casos porque pueden suponer mucha pérdida de información pero, si se opta por esta opción, podemos utilizar la función `dropna` de Pandas. Esta función altera el contenido de la colección sobre la que se aplica (además, debemos

especificar el parámetro `inplace=True`, habitual en Pandas para actualizar la colección original).

```
# Borrar filas con algún campo nulo
datos.dropna(inplace=True)

# Borrar filas que tengan en la columna "X" un valor nulo
datos.dropna(subset=['X'], inplace=True)
```

2.1.3. Aplicación al ejemplo

En lo que respecta a nuestro ejemplo, vamos a mostrar el porcentaje de nulos que hay en cada columna, volviendo a usar el código anterior. Vemos que las columnas con nulos son *age*, *gender* y *cardio*. En general son porcentajes muy bajos de nulos, y podríamos eliminar las filas afectadas con *dropna*, pero vamos a optar por diferentes alternativas

- Reemplazaremos los valores nulos de la edad (columna numérica) por la media de la columna
- Reemplazaremos los valores nulos del género (columna no numérica) por la moda de la columna
- Eliminaremos las filas que tengan el campo *cardio* nulo

```
# Reemplazo de edades nulas por la media
datos['age'] = datos['age'].fillna(datos['age'].mean())
# Reemplazo de géneros nulos por la moda
datos['gender'] = datos['gender'].fillna(datos['gender'].mode()[0])
# Eliminación de cardios nulos
datos = datos.dropna(subset=['cardio'])
```

2.2. Tipos de datos de las columnas

Vamos a analizar ahora el tipo de datos de cada columna para ver si es el adecuado. Esto se puede ver fácilmente con la propiedad `dtypes` del *dataset*. En nuestro ejemplo podemos ver que las columnas numéricas tienen un tamaño de 64 bits, cuando en realidad podrían ser de 32. También vemos que la columna *ap_lo* (presión sanguínea baja o diastólica) está marcada como tipo *object*, no numérica, lo que dificultará su uso en algunas operaciones. Cambiaremos su tipo también a *int32*:

```
datos = datos.astype({'age': 'int32', 'height': 'float32',
  'weight': 'float32', 'ap_hi': 'int32', 'ap_lo': 'int32'})
```

Nota

En el caso de que una columna tenga valores *NA* o *inf* es posible que no admita la conversión a tipo numérico. Habría que convertir o eliminar primero esos valores, para luego tratar el tipo de dato.

2.3. Gestión de anomalías (*outliers*)

Podemos definir una **anomalía** como la observación de algún dato que difiere significativamente del resto. Lo que se conoce en términos técnicos como un *outlier*. Por ejemplo, la siguiente lista contiene medidas de la altura de una puerta realizadas por diferentes personas:

```
2.1m, 2.3m, 4.5m, 2.2m, 2.4m
```

Claramente podemos detectar una anomalía en el tercer valor (*4.5m*) que difiere significativamente del resto. Los datos que extraemos del mundo real a menudo contienen este tipo de anomalías, y es importante detectarlas y gestionarlas aplicando alguna técnica conocida.

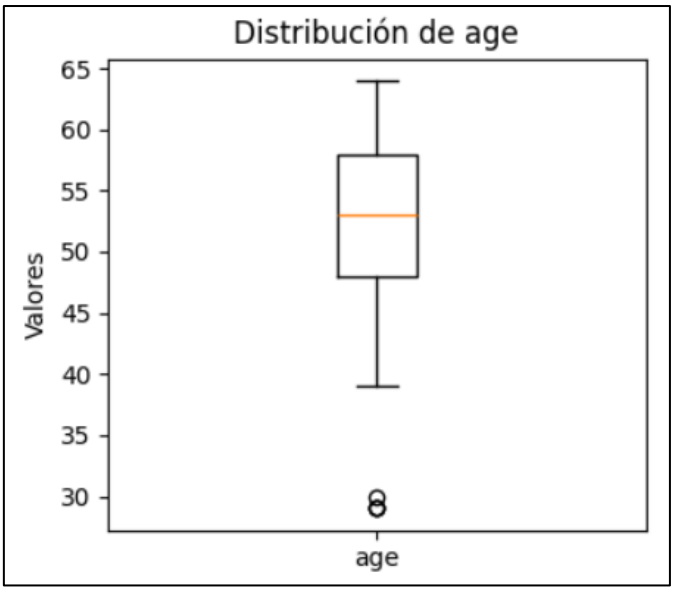
2.3.1. Detección gráfica de anomalías

Una primera aproximación a la detección de anomalías (en columnas numéricas) puede ser un gráfico de cajas (*box plot*), que represente la distribución de los valores de esa columna. Los puntos que queden más allá de los *bigotes* de las cajas serán las anomalías destacadas de ese campo.

Aplicado a nuestro ejemplo, podemos sacar el diagrama de cajas de las columnas *age*, *height*, *weight*, *ap_hi* y *ap_lo*:

```
categorias = ['age', 'height', 'weight', 'ap_hi', 'ap_lo']
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(8, 10))

axes = axes.flatten()
for i, var in enumerate(categorias):
    axes[i].boxplot(datos[var], tick_labels=[var])
    axes[i].set_title(f'Distribución de {var}')
    axes[i].set_ylabel('Valores')

for j in range(len(categorias), len(axes)):
    axes[j].axis('off')

plt.tight_layout()
plt.show()
```

Se puede ver fácilmente qué valores son anómalos en cada categoría. Aquí vemos el caso de la edad:



2.3.2. Detección de anomalías de forma matemática

Ver las anomalías en un gráfico puede ayudar a hacernos una idea de qué valores son los que hay que controlar en una categoría. Sin embargo, para poder detectarlas en el código y poderlas eliminar/modificar es necesario disponer a veces de algún método matemático. En este apartado proponemos varias alternativas.

Una estrategia habitual es la **detección de anomalías basada en la mediana**, que considera la mediana de un conjunto de valores como punto de referencia. Esta mediana simplemente es el valor del elemento central de un conjunto ordenado de valores. A partir de esta mediana, se define un cierto umbral alrededor, y cualquier valor que exceda ese umbral se considera una anomalía. Por ejemplo, dado un conjunto de valores en la variable `valores` y un umbral de 0.3, podríamos detectar qué valores son anómalos de este modo:

```
import numpy as np
import pandas as pd

valores = ... # Suponemos un conjunto de valores
mediana = np.median(valores)
umbral = 0.3
outliers = []
for elemento in valores:
    if abs(mediana - elemento) > umbral:
        outliers.append(elemento)
```

Otra alternativa es la **detección de anomalías basada en la media**. Consiste en utilizar la media de valores como punto de referencia, junto con la desviación típica. Esto evita tener que definir umbrales arbitrarios como en el caso anterior. Simplemente descartamos o anotamos como anomalías todos aquellos valores que excedan el rango de la media más/menos la desviación típica. Así quedaría para el ejemplo anterior:

```
media = np.mean(valores)
desviacion = np.std(valores)
outliers = []
for elemento in valores:
    if media - desviacion > elemento or media + desviacion < elemento:
        outliers.append(elemento)
```

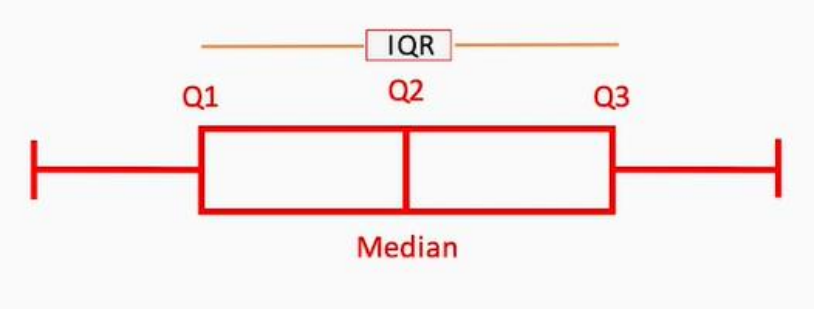
En el caso de que la desviación típica sea un umbral insuficiente, podemos optar por una variante conocida como **puntuación Z**, que consiste en definir como umbral un múltiplo de la desviación típica (puntuación Z) que indica a cuántas desviaciones típicas está un valor con respecto a la media:

$$Z = (\text{valor} - \text{media}) \,/\, \text{desviación típica}$$

Si la puntuación Z excede un cierto umbral razonable, consideramos al valor una anomalía. Aquí vemos cómo quedaría el ejemplo anterior usando una puntuación Z de 1.5:

```
media = np.mean(valores)
desviacion = np.std(valores)
outliers = []
for elemento in valores:
    z = abs(elemento - media) / desviacion
    if z > 1.5:
        outliers.append(elemento)
```

Finalmente, podemos optar por la técnica del cálculo del **rango intercuartílico** (*IQR*, *InterQuartile Range*). Un cuartil es una medida que divide el conjunto de valores en 4 áreas o intervalos. El rango intercuartil (IQR) es la zona que agrupa las dos áreas intermedias (segundo y tercer cuartil).



- El primer cuartil (Q1) deja a su izquierda el 25% de los valores
- El segundo cuartil (Q2) es siempre la mediana
- El tercer cuartil (Q3) deja a su izquierda el 75% de los valores

Según esta técnica, todos los valores que queden fuera de este rango IQR se consideran anomalías y deben descartarse. Podemos aplicar algún tipo de umbral para corregir esto e incluir valores que no se diferencien demasiado.

En Python, la función `percentile` de *NumPy* automáticamente ordena el vector de valores y calcula los cuartiles o percentiles que le digamos (es decir, los

valores que suponen un X% del total de valores). Así quedaría el código en el ejemplo anterior:

```
import numpy as np
import pandas as pd

# Nos interesan los cuartiles del 25% y 75%
Q1, Q3 = np.percentile(valores, [25, 75])
IQR = Q3 - Q1
outliers = []
for elemento in valores:
    if elemento < (Q1 - 1.5 * IQR) or elemento > (Q3 + 1.5 * IQR):
        outliers.append(elemento)
```

En este caso, hemos utilizado como umbral 1.5 veces el valor del IQR, descartando los valores que queden más allá, por arriba o por abajo.

2.3.3. Aplicación al ejemplo

¿Qué hacer con los datos que son *outliers*? Dependerá del problema en sí. Podemos descartarlos (quitarlos de la secuencia original de datos) o imputarlos (asignarles un valor alternativo, como por ejemplo el límite superior o inferior del umbral, entre otras opciones). En nuestro caso tomaremos las siguientes decisiones, en vista de los gráficos de caja obtenidos:

- Ignoraremos las anomalías de la edad, ya que corresponden a pacientes jóvenes (en torno a 30 años) pero podemos tenerlos en cuenta.
- Eliminaremos los registros de pacientes con altura superior a 230 cm
- Asignaremos un umbral inferior de 25 kg a todos los pacientes que pesen menos de esa cantidad (ignoraremos las anomalías de exceso de peso, porque pueden ser reales)
- Eliminaremos todas las presiones sanguíneas (*ap_hi* y *ap_lo*) que sean negativas o superiores a una puntuación Z de 2.

```
# Eliminar pacientes con altura superior a 230 cm
datos = datos[datos['height'] <= 230]

# Asignar 25 Kg a los pacientes que pesen menos de esa cantidad
datos['weight'] = datos['weight'].apply(
    lambda x: 25 if x < 25 else x
)
# Eliminar presiones sanguíneas negativas
datos = datos[(datos['ap_hi'] >= 0) & (datos['ap_lo'] >= 0)]

# Eliminar presiones sanguíneas superiores a Z = 2
# Paso 1: Obtenemos valores anómalos
media1 = datos['ap_hi'].mean()
media2 = datos['ap_lo'].mean()
desv1 = datos['ap_hi'].std()
desv2 = datos['ap_lo'].std()
outliers1 = []
```

```
outliers2 = []
Z = 2
for elemento in datos['ap_hi'].values:
    z = abs(elemento - media1) / desv1
    if z > Z:
        outliers1.append(elemento)
for elemento in datos['ap_lo'].values:
    z = abs(elemento - media2) / desv2
    if z > Z:
        outliers2.append(elemento)

# Paso 2: eliminar filas con outliers
datos = datos[~datos['ap_hi'].isin(outliers1)]
datos = datos[~datos['ap_lo'].isin(outliers2)]
```

Si consultamos la propiedad `shape` del *dataset* tras estos pasos de limpieza podremos ver que se han eliminado poco más de 1.000 registros de los 70.000 que teníamos inicialmente, lo que es una cantidad aceptable.

3. Ingeniería de características (*feature engineering*)

Ahora que ya hemos hecho una limpieza inicial de los datos de entrada, vamos a pasar al proceso de *feature engineering*. Entre otras cosas, este proceso consiste en:

- **Codificar las variables categóricas** (textuales) para darles un valor numérico, que es más apropiado para hacer operaciones matemáticas con estos valores de cara a obtener una predicción o resultado final
- **Elegir qué características** del conjunto de datos son más relevantes para el cálculo que queremos realizar. Es lo que se conoce como *feature selection*.
- **Escalar** los valores numéricos en un rango homogéneo, para que no haya valores con una magnitud superior a la de otros.

3.1. Codificación de variables categóricas

En muchos datasets podremos encontrar datos alfanuméricos, también llamados *categorizados*. Por ejemplo, nombres de ciudades, o calificaciones alfabéticas ("buena", "muy buena"...). Estos datos categorizados no suelen formar parte de algunos análisis por la imposibilidad de hacer operaciones con ellos: no podemos sacar la media de unas ciudades, por ejemplo, o multiplicar una valoración "buena" por un coeficiente numérico.

Para evitar esto y hacer que esos datos también formen parte del problema a resolver, lo que se suele hacer es codificarlos en datos numéricos. Existen para ello distintas estrategias; comentaremos aquí un par de ellas:

- Codificación de etiquetas (*label encoding*)
- Codificación *one hot*

3.1.1. Codificación de etiquetas (*label encoding*)

El etiquetado de datos categóricos consiste en asignar un valor numérico a cada posible valor categórico de una serie de datos. Por ejemplo, imaginemos una tabla como la siguiente, que podría almacenar algunos datos de participantes de un club deportivo:

Nacionalidad	Edad	Peso	Socio
España	34	88.4	Si
Portugal	38	95.6	Si
España	30	90.2	No
Francia	40	96.7	No
Portugal	37	99.3	Si
España	32	82.4	Si

Podemos construir el *dataframe* equivalente de este modo:

```
import pandas as pd

datos = { 'Nacionalidad': ['España', 'Portugal', 'España',
                           'Francia', 'Portugal', 'España'],
          'Edad': [34, 38, 30, 40, 37, 32],
          'Peso': [88.4, 95.6, 90.2, 96.7, 99.3, 82.4],
          'Socio': ['Si', 'Si', 'No', 'No', 'Si', 'Si']
        }

df = pd.DataFrame(datos)
```

Si quisiéramos establecer una conexión entre los datos de las personas y si son socios o no, no podríamos hacer nada con la nacionalidad, porque es categórica. Tendríamos que asignarle un valor numérico equivalente para poder, por ejemplo, multiplicarla por un coeficiente en una ecuación y establecer así una correlación entre la nacionalidad y la probabilidad de ser socio o no.

Para ello, el primer paso que tendremos que dar es definir el tipo de dato de la(s) columna(s) afectada(s) como *category* en lugar del tipo por defecto *object* que les asigna Pandas:

```
df['Nacionalidad'] = df['Nacionalidad'].astype('category')
```

Después, podemos obtener los códigos que automáticamente se han asignado a cada categoría con las propiedades `cat.codes`, e incluso crear otra columna con ello:

```
df['Cod_Nacionalidad'] = df['Nacionalidad'].cat.codes
```

También podemos aplicar este tipo de codificación a la columna *Socio*, obteniendo los valores alternativos 0 y 1:

```
df['Socio'] = df['Socio'].astype('category')
df['Cod_Socio'] = df['Socio'].cat.codes
```

Obtendremos como resultado algo así:

```
  Nacionalidad  Edad  Peso Socio  Cod_Nacionalidad  Cod_Socio
0       España    34  88.4    Si                 0          1
1     Portugal    38  95.6    Si                 2          1
2       España    30  90.2    No                 0          0
3      Francia    40  96.7    No                 1          0
4     Portugal    37  99.3    Si                 2          1
5       España    32  82.4    Si                 0          1
```

3.1.2. Codificación *one hot*

La codificación de etiquetas anterior puede resultar problemática en algunas ocasiones. Al asignar un valor numérico diferente a cada categoría de un conjunto, sin quererlo, se establece un orden. Así, para el ejemplo anterior, si por ejemplo obtenemos que *España* tiene el valor 0, *Portugal* el 1 y *Francia* el 2, si usamos esas codificaciones en un sistema de *machine learning* se podría llegar a deducir que *Francia* es mayor o mejor que *España* porque 2 > 0.

Como ése no suele ser el propósito, una alternativa consiste en utilizar la codificación *one hot*. Una de las principales características de este tipo de codificación es que no puede haber ninguna relación de orden entre los elementos codificados, porque se codifican en distintas columnas, como veremos a continuación.

Para aplicar codificación *one hot* con Pandas sobre una columna determinada, podemos emplear el método `get_dummies`, indicando el *data frame* con los datos, las columnas que queremos codificar (en el parámetro `columns`, en forma de lista), y el prefijo que queremos darle a cada nueva columna que se genera. Apliquémoslo a la columna de nacionalidad anterior, de este modo:

```
df = pd.get_dummies(df, columns=['Nacionalidad'], prefix='Nac')
```

La función `get_dummies` devuelve un *data frame* donde se sustituyen las columnas indicadas por las nuevas. Esto elimina la columna categórica *Nacionalidad* y la reemplaza por las columnas *one hot* que se generan. Obtendremos este resultado:

```
   Edad  Peso Socio  Nac_España  Nac_Francia  Nac_Portugal
0    34  88.4    Si           1            0             0
1    38  95.6    Si           0            0             1
2    30  90.2    No           1            0             0
3    40  96.7    No           0            1             0
4    37  99.3    Si           0            0             1
5    32  82.4    Si           1            0             0
```

Si no queremos perder la columna categórica por algún motivo, podemos usar la función de este otro modo: lo que hacemos es generar las columnas *one hot* aparte (sólo pasándole la columna o columnas a codificar, en lugar de todo el *data frame*), y luego enlazarlas con `join` al *data frame* original:

```
columnas_one_hot = pd.get_dummies(df['Nacionalidad'],
    columns=['Nacionalidad'], prefix='Nac')
```

```
df = df.join(columnas_one_hot)
```

Obtenemos este otro resultado:

```
  Nacionalidad  Edad  Peso Socio  Nac_España  Nac_Francia  Nac_Portugal
0       España    34  88.4    Si           1            0             0
1     Portugal    38  95.6    Si           0            0             1
2       España    30  90.2    No           1            0             0
3       Francia   40  96.7    No           0            1             0
4     Portugal    37  99.3    Si           0            0             1
5       España    32  82.4    Si           1            0             0
```

Como podemos ver, se generan tantas columnas como posibles valores tiene la categoría, y así se pone a 1 la columna a la que pertenece, y a 0 el resto.

3.1.3. Cuando usar cada tipo de codificación

Existen otros tipos de codificación de datos categóricos que no hemos mencionado aquí, pero el uso de una u otra técnica obedecerá a ciertas premisas en el problema y los datos que estemos manejando.

En general, usaremos codificación *one hot* cuando no haya ninguna relación de orden entre las categorías, y usaremos *label encoding* cuando sí pueda haber cierta relación de orden o preferencia entre los valores categóricos. Para la columna *Socio* del caso anterior podríamos elegir cualquiera de las dos opciones, ya que sólo toma dos valores (0 o 1), e incluso podríamos querer decir que ser socio (1) es "mejor" que no serlo (0), según el problema. También usaremos *label encoding* para valoraciones, como en el siguiente ejemplo, donde sí interesa establecer un orden o gradación entre las categorías:

Valoración	**Codificación**
Mala	0
Regular	1
Buena	2
Muy buena	3

Hay que tener en cuenta que, en este caso, los *label encoders* que hemos utilizado antes no tienen por qué asignar una numeración "correcta" para nuestros intereses, y tendríamos que hacerlo manualmente. Podemos crear una nueva columna paralela a la original *Valoracion*, llenarla con un valor inicial (cero, por ejemplo), y luego poner el valor a 1 en el caso de valoraciones *Regular*, 2 para valoraciones *Buena*, etc.

```
# Creamos columna Codificacion a partir de la Valoracion
datos['Codificacion'] = datos['Valoracion'].replace(
    ['Mala', 'Regular', 'Buena', 'Muy buena'], [0, 1, 2, 3])
```

3.1.4. Aplicación al ejemplo

Volvamos a nuestro ejemplo de datos de salud. En el *dataset* tenemos varias columnas categóricas. Aquí indicaremos lo que vamos a hacer con ellas:

- La columna *gender* toma los valores *M* (masculino) o *F* (femenino). Crearemos una codificación *one hot* para distinguir con ceros y unos la pertenencia a uno u otro grupo.
- Las columnas *cholesterol* y *gluc* pueden tomar los valores *Normal*, *Above Normal* y *Well Above Normal*. En este caso sí nos interesa que haya una gradación o relación de orden, ya que un nivel de colesterol *Normal* es mejor que uno *Above Normal*, y éste a su vez es mejor que el *Well Above Normal*. Para estas columnas usaremos *label encoding*.
- Las columnas *smoke*, *alco*, *active* y *cardio* tienen valores *Yes/No*, que podemos codificar indistintamente como *one hot* o *label encoding*, ya que el resultado será el mismo (valores 1/0). Nos aseguraremos, en cualquier caso, que el valor *Yes* equivalga a 1 y el *No* a 0.

```
# Codificación "one hot" de la columna "gender"
datos = pd.get_dummies(datos, columns=['gender'], prefix='Gender')

# Codificaciones "label encoding" de "cholesterol" y "gluc"
datos['cholesterol'] = datos['cholesterol'].replace(['Normal', 'Above
Normal', \
    'Well Above Normal'], [0, 1, 2])
datos['gluc'] = datos['gluc'].replace(['Normal', 'Above Normal', \
    'Well Above Normal'], [0, 1, 2])

# Codificaciones binarias de columnas SI/NO
datos['smoke'] = datos['smoke'].replace(['No', 'Yes'], [0, 1])
datos['alco'] = datos['alco'].replace(['No', 'Yes'], [0, 1])
datos['active'] = datos['active'].replace(['No', 'Yes'], [0, 1])
datos['cardio'] = datos['cardio'].replace(['No', 'Yes'], [0, 1])

# Vemos cómo queda el dataset
print(datos.head())
```

En algunas ocasiones, la codificación *one hot* genera información redundante. En nuestro caso, tenemos ahora dos columnas *Gender_F* y *Gender_M* que nos indican si un paciente es hombre o mujer. Nos bastaría con una de las dos, ya que si no pertenece a un género automáticamente se le asigna el otro. Así que eliminamos, por ejemplo, la columna *Gender_M* del estudio.

```
datos.drop('Gender_M', axis=1, inplace=True)
```

3.2. Selección de características (*feature selection*)

Vamos a abordar en este apartado la selección de las características o columnas más relevantes para nuestro estudio. Saber elegir las características correctas o adecuadas para un problema puede repercutir en múltiples ventajas, tales como la reducción del proceso de aprendizaje, el ahorro de memoria de almacenamiento o la simplificación del modelo a considerar. Además, ayuda a

evitar la codependencia o *colinearidad* entre atributos. Es decir, si un atributo depende fuertemente de otro, quizá analizando uno de los dos sea suficiente, y podemos descartar el segundo.

Existen distintas estrategias que nos pueden ayudar a tomar esta decisión. Por ejemplo, podemos entrenar a nuestro sistema con diferentes combinaciones de características, y ver en cuáles se obtienen resultados significativamente diferentes. En nuestro caso vamos a optar por analizar la correlación entre los valores, a través de un **mapa de calor de correlación** (*correlation heatmap*) gracias a *Pandas* y *Seaborn*.

Este mapa nos mostrará de forma gráfica las dependencias entre las diferentes variables involucradas: una correlación alta (cercana a 1) entre dos variables indicará que cuando una aumenta la otra también lo hace, y esto puede significar que una influye mucho en el valor de la otra (y es determinante para calcularlo), o bien que las dos tienen un comportamiento similar (y podemos prescindir de una de ellas). También tenemos que tener en cuenta la correlación inversa (cercana a -1) y ver qué hacer en esos casos.

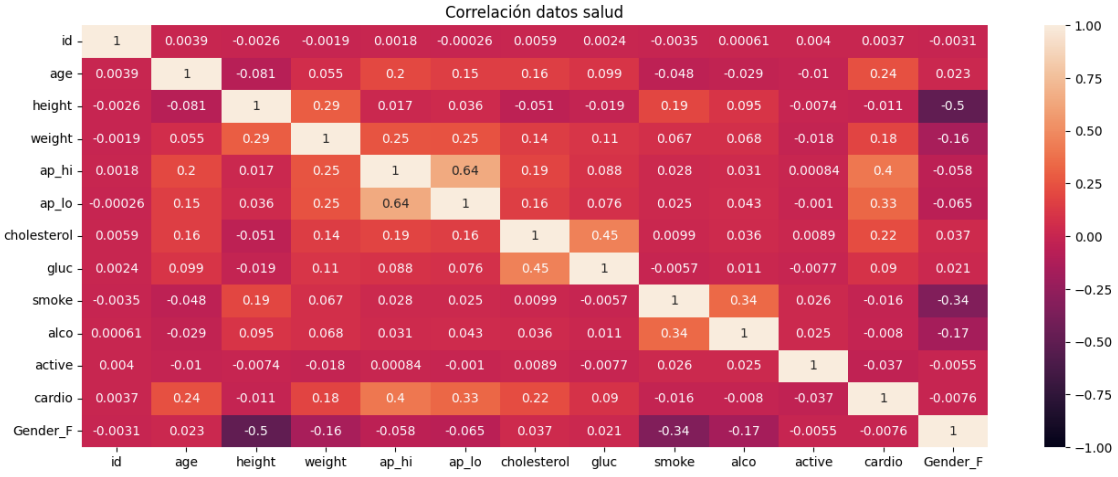
En definitiva, podemos eliminar características del conjunto por dos motivos:

- Porque sean **redundantes** (muy correlacionadas con otras que ya vamos a incluir)
- Porque sean **irrelevantes** (no afecten al resultado que se quiere obtener)

Construimos un mapa de calor de correlación sobre nuestro *dataset*, de este modo:

```
plt.figure(figsize=(16, 6))
heatmap = sns.heatmap(datos.corr(), vmin=-1, vmax=1, annot=True)
heatmap.set_title('Correlación datos salud')
plt.show()
```

Esto generará un gráfico de tipo mapa de calor, con la tabla de correlaciones:



En base a este mapa de correlación, podemos ir a la columna *ap_hi*, que es la que corresponde a nuestra variable objetivo, y ver qué otras variables son las que más influyen en su valor. Podemos ver que son la presión sanguínea baja (*ap_lo*), si padece o no problemas cardíacos (columna *cardio*), el peso (*weight*), la edad (*age*) y si tiene o no colesterol. El resto de valores son muy cercanos a 0, es decir, no parecen muy relacionados con la presión sanguínea alta y se pueden eliminar.

Además, si observamos las variables que hemos seleccionado, es decir, *ap_lo*, *cardio*, *weight*, *age* y *cholesterol* podremos ver que no existe una correlación fuerte, lo que nos permitiría eliminar alguna de ellas y quedarnos con un conjunto más reducido.

Vamos entonces a quedarnos únicamente con estas columnas de nuestro *dataset*:

```
columnas_relevantes = ['age', 'weight', 'ap_lo', 'cholesterol', 'cardio']
datos = datos[columnas_relevantes + [valor_objetivo]]
datos.head()
```

3.3. Escalado de características (*feature scaling*)

El escalado de características o *feature scaling* es una herramienta muy habitual en el procesamiento de datos. Consiste en dejar un conjunto de valores en un rango común o delimitado. Esto es bastante útil, por ejemplo, en el campo de las redes neuronales. Imaginemos una red que toma imágenes como datos de entrada. Los valores de los colores de esas imágenes pueden estar definidos en distintos rangos, según el formato de la imagen (0 a 255 para imágenes en escala de grises, o valores mayores para algunos modelos de color). En este caso, podría interesar que los valores de los colores de los píxeles estuvieran normalizados en un rango de 0 a 1 para un mejor tratamiento.

Otro ejemplo quizá más ilustrativo. Imaginemos que tenemos datos de clientes de una entidad bancaria. Guardamos su edad, sus ingresos anuales, provincia, etc:

Edad	**Ingresos**	**Provincia**
34	28000	Sevilla
40	30000	Toledo
54	32000	Oviedo
43	38000	Badajoz

Si queremos determinar cómo de parecidos o diferentes son dos individuos en base a ciertos datos de entrada (por ejemplo, edad y e ingresos), esto se puede conseguir calculando la "distancia" entre estos dos individuos en base a esos parámetros:

raiz_cuadrada((edad1 - edad2)^2 + (ingresos1 - ingresos2)^2)

El problema aquí lo encontramos en que ingresos y edad se mueven en rangos de valores diferentes, por lo que a la hora de determinar la diferencia entre dos individuos van a ser mucho más determinantes los ingresos, por ser valores mucho más altos. Así, una diferencia de ingresos de apenas 500 euros va a marcar mucho más la diferencia entre individuos que una diferencia de edad de 30 años. Para evitar este problema, podemos escalar los datos a un rango común (por ejemplo, que ambas columnas se muevan en un rango de 0 a 1). Vamos a ver cómo se hace.

3.3.1. Escalado por normalización

Una primera estrategia de escalado es la normalización, también llamada **escalado *min-max***, que deja todos los valores del conjunto en el rango de 0 a 1, aplicando la siguiente fórmula a cada valor:

val_normalizado = (val_actual - val_min) / (val_max - val_min)

Imaginemos un conjunto de datos como este:

```
import pandas as pd

datosIniciales = {'Nombre': ['Juan', 'Ana', 'Mario', 'Laura'],
    'Edad': [70, 40, 30, 26],
    'Sueldo': [2800, 1200, 1750, 1420]}
datos = pd.DataFrame(datosIniciales)
```

Podemos aplicar el escalado *min-max* a la colección, aplicándolo a cada columna afectada:

```
datos['Edad'] = (datos['Edad'] - datos['Edad'].min()) / \
    (datos['Edad'].max() - datos['Edad'].min())

datos['Sueldo'] = (datos['Sueldo'] - datos['Sueldo'].min()) / \
    (datos['Sueldo'].max() - datos['Sueldo'].min())
```

En el caso de que todas las columnas de la colección sean numéricas y queramos normalizarlas, podemos aplicar una sola fórmula a todo el *data frame* (no es el caso de este ejemplo):

datos = (datos - datos.min()) / (datos.max() - datos.min())

3.3.2. Escalado por estandarización

Una segunda alternativa consiste en calcular la media y desviación típica del conjunto de datos a tratar. El valor normalizado se calcula entonces como:

val_normalizado = (val_actual - media) / desviación

En este caso, el rango de valores ya no va de 0 a 1, y admite valores negativos, pero al menos está acotado a un rango más controlado. Así quedaría en nuestro ejemplo anterior:

```
datos['Edad'] = (datos['Edad'] - datos['Edad'].mean()) / \
    datos['Edad'].std()

datos['Sueldo'] = (datos['Sueldo'] - datos['Sueldo'].mean()) / \
    datos['Sueldo'].std()
```

3.3.3. Aplicación al ejemplo

Vamos a aplicar el escalado a nuestro ejemplo. Las columnas codificadas como *one hot* no es necesario escalarlas, puesto que ya están acotadas en un rango 0-1. Las columnas codificadas como *label encoding* podríamos escalarlas si quisiéramos en el caso de que las etiquetas tuvieran valores más altos. Pero en nuestro caso sólo van del 0 al 2. Así que nos centraremos en la edad, el peso y la presión sanguínea baja.

Aplicaremos el escalado por estandarización.

```
# Escalamos la edad
datos['age'] = (datos['age'] - datos['age'].mean()) / datos['age'].std()

# Escalamos el peso
datos['weight'] = (datos['weight'] - datos['weight'].mean()) /
datos['weight'].std()

# Escalamos la presión baja
datos['ap_lo'] = (datos['ap_lo'] - datos['ap_lo'].mean()) /
datos['ap_lo'].std()
```

4. Análisis exploratorio de datos

El análisis exploratorio de datos (*EDA*) es una etapa que suele darse simultáneamente a las dos anteriores, y nos permite sacar ciertas conclusiones sobre los datos con los que estamos trabajando. Su objetivo principal es comprender las características y patrones de los datos, identificar relaciones, detectar valores atípicos y evaluar posibles problemas como datos faltantes. Este proceso guía la preparación de los datos y la selección de modelos.

De hecho, ya hemos hecho uso de dicho proceso cuando hemos mostrado los diagramas de cajas de las columnas numéricas para conocer la distribución de valores y ver posibles anomalías, y también cuando hemos obtenido el mapa de calor de correlación para conocer qué parámetros eran más relevantes para definir el valor objetivo. También cuando hemos empleado el método `describe` para obtener los datos estadísticos de la columna objetivo *ap_hi*, y conocer sus valores máximos, mínimos, media, etc.

Además de todo eso, podemos obtener otras representaciones numéricas y gráficas, como por ejemplo un histograma con la distribución de valores de la columna objetivo *ap_hi*, o también gráficos de dispersión (*scatter plots*) que muestren la dependencia de cada columna relevante con la variable objetivo.

Podemos mostrar todo esto en una matriz de 3 filas y 2 columnas, por ejemplo:

```
fig, ax = plt.subplots(nrows=3, ncols=2, figsize=(12, 10))
for i in range(0, len(columnas_relevantes)):
  fila = i // 2
  columna = i % 2
  ax[fila, columna].scatter(x = datos[columnas_relevantes[i]],
    y = datos[valor_objetivo])
  ax[fila, columna].set_title(columnas_relevantes[i] + " frente a " +
valor_objetivo)
ax[2, 1].hist(datos[valor_objetivo], bins=10)
ax[2, 1].set_title("Distribución de valores de " + valor_objetivo)
```

El EDA busca responder preguntas clave sobre los datos, basándose en técnicas estadísticas:

- ¿Cómo están distribuidas las variables?
- ¿Existen relaciones significativas entre las variables?
- ¿Hay datos faltantes, valores atípicos o errores?
- ¿Cuál es el nivel de variabilidad en los datos?

Un análisis detallado permite tomar decisiones informadas sobre el preprocesamiento y la modelización.

Pasos principales del análisis exploratorio

1. Comprensión inicial de los datos

- **Estructura de los datos**: número de filas y columnas, tipos de datos, y resumen básico.
- **Datos faltantes**: Identificación y porcentaje de valores faltantes por variable.
- **Valores únicos**: Verificación de categorías o clases.

2. Análisis estadístico univariado: el análisis estadístico univariado examina cada variable de forma individual. Las medidas comunes son:

- **Tendencia central**: Media, mediana, moda.
- **Dispersión**: Desviación estándar, varianza, rango.
- **Forma de la distribución**: Asimetría (skewness) y curtosis.

2.3. Análisis estadístico bivariado: este análisis examina la relación entre dos variables y cómo una afecta a la otra. **Ejemplo de técnicas:**

- **Correlación (variables numéricas)**: La correlación de Pearson mide la relación lineal.
- **Tablas de contingencia (variables categóricas)**: Evalúa la relación entre categorías.
- **Pruebas estadísticas**: Por ejemplo, la prueba t para comparar medias de dos grupos.

2.4. Identificación de valores atípicos: los valores atípicos pueden influir significativamente en los modelos. Métodos comunes para detectarlos:

- **Boxplots**: Visualización gráfica de valores atípicos.
- **IQR (Rango Intercuartil)**: Detecta valores fuera de 1.5 veces el rango intercuartil.
- **Z-score**: Calcula cuán lejos está un valor de la media.

2.5. Análisis de distribuciones: comprender las distribuciones ayuda a elegir modelos y técnicas de preprocesamiento.

- **Distribuciones normales**: Son ideales para muchas técnicas estadísticas.
- **Distribuciones sesgadas**: Pueden requerir transformaciones (logaritmo, raíz cuadrada, etc.).

3. Visualización del análisis estadístico: las visualizaciones son **clave** y absolutamente **fundamentales** para complementar el análisis estadístico. Algunas herramientas útiles incluyen:

- **Gráficos univariados**: Histogramas, boxplots.
- **Gráficos bivariados**: Scatter plots, heatmaps de correlación.
- **Distribuciones categóricas**: Gráficos de barras o proporciones.

4. Conclusión del EDA

Tras realizar el análisis exploratorio de datos, el objetivo es resumir los hallazgos clave, identificar posibles problemas (valores atípicos, datos faltantes, etc.) y planificar los próximos pasos, como la ingeniería de características, transformación de datos o selección de variables. El análisis estadístico proporciona una base sólida para construir modelos predictivos más robustos y precisos.

5. Desarrollo de un modelo supervisado de regresión

Para terminar con este ejemplo vamos a construir un modelo de árbol de decisión que entrene con el conjunto de datos que hemos definido, y mediremos su precisión final (*accuracy*).

5.1. Definición de la métrica

En primer lugar, vamos a definir la métrica, es decir, una función que determine cómo de bien o mal estamos haciendo nuestra tarea (estimar la presión sistólica). Emplearemos para ello como medida el error absoluto medio (MAE).

```
def metrica(valores_reales, valores_predichos):
    return mean_absolute_error(valores_reales, valores_predichos)
```

En este caso el MAE puede resultar adecuado para obtener el error cometido, y nos devolverá cómo nos equivocamos (en promedio) con las estimaciones respecto a los valores reales, calculando la diferencia en valor absoluto entre unas y otras, y obteniendo la media de esos errores.

5.2. Definición de conjuntos de entrenamiento y test

En todo proceso de estimación es necesario disponer de un conjunto de datos con los que "practicar" y entrenar al sistema, y otro conjunto de datos, con resultados conocidos, con los que determinar si el sistema realmente ha aprendido a hacer las cosas adecuadamente o no. Este segundo conjunto se suele denominar datos de validación.

Usaremos la función `train_test_split` de *sklearn* para dividir automáticamente nuestro conjunto original de datos en dos partes: una para entrenamiento y otra para validación.

```
# Indicamos que el 20% de los datos son para test
# random_state sirve para que siempre se elijan los mismos datos para test.
df_train, df_val = train_test_split(datos, test_size=0.2, random_state=12)
```

5.3. Definición del modelo

Vamos ahora a definir un árbol de decisión simple para evaluar cómo de bien o mal estima esta presión sistólica con estos datos de entrenamiento y test:

```
modelo_arbol = DecisionTreeRegressor(random_state=12, max_depth=7,
    min_samples_split=10)
modelo_arbol.fit(df_train[columnas_relevantes], df_train[valor_objetivo])
predicciones = modelo_arbol.predict(df_val[columnas_relevantes])
error_val = metrica(df_val[valor_objetivo], predicciones)
print(f'Métrica para datos de validación: {error_val}')
```

Si hemos seguido los pasos de este documento probablemente obtengamos un error promedio en torno a 7.6, es decir, 7 puntos de diferencia entre la presión predicha y la real. Es algo mejorable, pero nos da una idea, en definitiva, de los pasos que se deben seguir para preparar un conjunto de datos de cara a la definición de un modelo.

5.4. Algunas consideraciones finales

Hemos visto durante estos últimos apartados algunas estrategias de escalado y codificación de datos. Nos puede surgir la duda de cuándo aplicar unas u otras, y sobre qué datos de entrada en concreto. Realmente no hay una respuesta universal a esta pregunta, porque va a depender mucho del conjunto de datos de entrada y de lo que queramos hacer con ellos (regresión, clasificación, etc). Pero podemos dar algunas pautas generales a tener en cuenta:

- Normalmente escalaremos columnas que tengan rangos de valores muy dispares entre sí, para aunarlas todas en un rango común (usando normalización o estandarización, como queramos)
- Los valores *one hot* pueden escalarse o no, hay razones buenas en ambos casos, pero hay que tener en cuenta que, si los escalamos, perderemos esa noción de "pertenencia a una categoría" que nos dan los ceros y unos de cada columna, ya que pasarán a tener otros valores. Es habitual, por tanto, no escalarlos en muchos problemas.
- Los valores a predecir (típicamente conocidos como columna *y* o variable dependiente) también podemos decidir si escalarlos o no, dependiendo del problema: si estamos en un problema de clasificación, lo normal será no escalarlos, para que se asigne cada registro a una categoría igual a las que teníamos de entrada. Si estamos en un problema de regresión y hemos escalado los parámetros de entrada, es posible que también nos interese escalar el valor de salida (eje vertical).

6. Métricas de validación de modelos de regresión

Además del MAE, usado anteriormente en el ejemplo, para valorar la calidad de los modelos de regresión tenemos que emplear unas métricas determinadas. Vamos a revisar las más importantes.

1. Error Absoluto Medio (MAE - Mean Absolute Error)

El MAE mide el promedio de las diferencias absolutas entre las predicciones y los valores reales. Es fácil de interpretar porque utiliza las mismas unidades que la variable objetivo, también es robusto frente a outliers.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

2. Error Cuadrático Medio (MSE - Mean Squared Error)

Penaliza los errores más grandes más que el MAE, ya que los eleva al cuadrado. Esto puede ser útil si los errores grandes son especialmente importantes en tu problema. Cuidado con su interpretación ya que son unidades cuadradas.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

3. Raíz del Error Cuadrático Medio (RMSE)

Es la raíz cuadrada del MSE, por lo que devuelve los errores en las mismas unidades que la variable objetivo.

4. Coeficiente de Determinación (R^2)

Mide qué tan bien las predicciones se ajustan a los datos reales. Va de 0 a 1 (en casos ideales) y puede ser negativo si el modelo es peor que simplemente usar la media de los datos.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Veamos cómo invocarlos (luego pueden ser mostrados en pantalla):

```
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

mae = mean_absolute_error(y_true, y_pred)
mse = mean_squared_error(y_true, y_pred)
r2 = r2_score(y_true, y_pred)
# Reflexiona: ¿por qué no sale una función para el RMSE?
```

7. Modelos supervisados

1. **Regresión logística** (Logistic Regression): modelo para clasificación binaria (o multiclase) basado en probabilidades logísticas. Hiperparámetros importantes:

- **penalty**: Tipo de regularización a aplicar ('l1', 'l2', 'elasticnet').
- **C**: Inverso de la regularización; valores más altos implican menos regularización.
- **solver**: Algoritmo para la optimización ('lbfgs', 'liblinear', etc.).

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(penalty='l2', C=1.0, solver='lbfgs',
max_iter=1000)
model.fit(X_train, y_train)
```

2. **Árboles de decisión** (Decision Tree): modelo no paramétrico que divide iterativamente el espacio de las variables según condiciones óptimas. Útil para regresión y clasificación. Hiperparámetros importantes:

- **max_depth**: Profundidad máxima del árbol.
- **min_samples_split**: mínimo de muestras para dividir un nodo.
- **criterion**: Función para medir la calidad de la división ('gini', 'entropy' para clasificación; 'squared_error', 'friedman_mse' para regresión).

```
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor

model = DecisionTreeClassifier(max_depth=5, criterion='gini',
min_samples_split=10)
model.fit(X_train, y_train)
```

3. **Bosques aleatorios** (Random Forest): ensamble de árboles de decisión que combina predicciones de múltiples árboles entrenados en diferentes subconjuntos del dataset. Funciona para regresión y clasificación. Hiperparámetros importantes:

- **n_estimators**: Número de árboles en el bosque.
- **max_features**: Número de características consideradas en cada división.
- **bootstrap**: Si se utilizan muestras con reemplazo para entrenar cada árbol.

```
from sklearn.ensemble import RandomForestClassifier,
RandomForestRegressor

model = RandomForestClassifier(n_estimators=100, max_depth=7,
bootstrap=True)
model.fit(X_train, y_train)
```

4. **K-Vecinos más cercanos** (K-Nearest Neighbors, KNN): modelo basado en la distancia entre puntos para predecir valores o clasificaciones según los vecinos más cercanos. Hiperparámetros importantes:

- **n_neighbors**: Número de vecinos considerados.
- **metric**: Métrica para calcular la distancia ('euclidean', 'manhattan', etc.).
- **weights**: Ponderación de los vecinos ('uniform', 'distance').

```
from sklearn.neighbors import KNeighborsClassifier,
KNeighborsRegressor

model = KNeighborsClassifier(n_neighbors=5, metric='euclidean',
weights='uniform')
model.fit(X_train, y_train)
```

5. **Máquina de Vectores de Soporte** (SVM): busca un hiperplano óptimo que separe los datos en clasificación o que ajuste los valores en regresión. Hiperparámetros importantes:

- **kernel**: Tipo de transformación del espacio de datos ('linear', 'rbf', 'poly').
- **C**: Regularización; valores más altos dan menos regularización.
- **gamma**: Influencia de un solo punto para el kernel ('scale', 'auto').

```
from sklearn.svm import SVC, SVR

model = SVC(kernel='rbf', C=1.0, gamma='scale')
model.fit(X_train, y_train)
```

6. **XGBoost**: variante optimizada de Gradient Boosting. Es robusto frente a valores atípicos y escalable para grandes datasets. Hiperparámetros importantes:

- **max_depth**: Profundidad máxima de los árboles.
- **eta** (learning_rate): Tasa de aprendizaje.
- **gamma**: Reducción mínima en la pérdida requerida para dividir un nodo.

```
from xgboost import XGBClassifier

model = XGBClassifier(max_depth=5, learning_rate=0.1,
n_estimators=100, gamma=0.3)
model.fit(X_train, y_train)
```

Estos modelos podrían proporcionar diferentes perspectivas sobre la relación entre las características de salud y la presión sanguínea alta, permitiendo comparar su rendimiento con el modelo inicialmente empleado en el ejemplo.

8. Mejora de los modelos

Hay situaciones que complican la interpretación de los resultados de un modelo, la calidad del mismo, el rendimiento y muchas otras problemáticas. Para luchar con todo ello existen una serie de técnicas importantes a conocer. Veamos las más significativas.

8.1. Datasets desbalanceados: sobremuestreo y submuestreo

Un dataset desbalanceado puede causar varios problemas en el desarrollo de modelos de machine learning. Los algoritmos suelen estar diseñados para maximizar la precisión global, lo que puede llevar a un **sesgo hacia la clase mayoritaria**, ignorando la clase minoritaria y reduciendo su capacidad para identificarla correctamente.

Esto genera métricas engañosas, como una precisión aparentemente alta, pero con un bajo desempeño en métricas clave como el recall o el F1-score de la clase minoritaria. Además, aumenta el riesgo de **falta de generalización**, ya que el modelo no aprende patrones útiles para la clase minoritaria, y en casos extremos puede provocar overfitting a la clase mayoritaria. Estos problemas tienen un impacto ético y práctico, especialmente en aplicaciones críticas como detección de fraudes o diagnósticos médicos, donde fallar en la clase minoritaria puede tener consecuencias significativas.

Por ejemplo, supongamos que tienes un dataset con transacciones bancarias, donde:

- 98% de las transacciones son legítimas (Clase 0).
- 2% son fraudulentas (Clase 1).

El problema es que el modelo podría aprender a predecir siempre **Clase 0** (legítima), logrando una precisión del **98%**, debido a la gran cantidad de muestras de esa clase frente a la otra, produciendo un evidente sesgo, pero fallando completamente en detectar fraudes. Los impactos son:

1. **Sesgo**: El modelo ignora completamente la clase minoritaria (fraudes).
2. **Métricas engañosas**: Aunque la precisión es alta (98%), el modelo tiene un Recall del **0%** para la clase 1.
3. **Problema principal**: No puede detectar fraudes, lo que lo hace inútil para su propósito.

Para poder mitigar este problema podemos aplicar técnicas como:

1. **Oversampling** con SMOTE para aumentar la cantidad de ejemplos de fraudes.
2. **Undersampling** para reducir ejemplos de transacciones legítimas.
3. Usar métricas más adecuadas, como **F1-score** o **ROC-AUC**, para evaluar el modelo.

```python
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# Datos simulados
X, y = ...  # Transacciones y etiquetas (0 = No fraude, 1 = Fraude)

# SMOTE para balancear
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Entrenamiento y evaluación del modelo
clf = RandomForestClassifier(random_state=42)
clf.fit(X_resampled, y_resampled)
y_pred = clf.predict(X)
print(classification_report(y, y_pred))
```

Las técnicas de **oversampling** y **undersampling** se utilizan para abordar el problema de conjuntos de datos desbalanceados, en los que una o más clases tienen significativamente menos ejemplos que otras. Estas técnicas ajustan la distribución de clases para mejorar el rendimiento de los modelos supervisados. En **scikit-learn**, estas técnicas se implementan a menudo mediante la biblioteca **imbalanced-learn (imblearn)**.

Además de SMOTE, también podemos emplear como alternativas:

1. **Random Oversampling**: duplica las muestras de la clase minoritaria al azar.
2. **ADASYN**: similar a SMOTE, pero genera ejemplos basados en la densidad de las muestras minoritarias.

```python
from imblearn.over_sampling import RandomOverSampler, ADASYN
oversampler = RandomOverSampler(random_state=42)
adasyn = ADASYN(random_state=42)
```

Si no queremos depender de otras librerías, sklearn también nos ofrece una alternativa para realizar “resampling”, duplicando las muestras de la clase minoritaria o seleccionando aleatoriamente una parte de las muestras de la clase mayoritaria. Veamos un ejemplo conciso:

```python
from sklearn.utils import resample

# Supongamos que separamos clases mayoritaria y minoritaria
df_major = df[df['target'] == 0]
df_minor = df[df['target'] == 1]

# Ahora hacemos oversampling de la clase minoritaria
df_minor_oversampled = resample(df_minor,
replace=True,n_samples=len(df_major), random_state=42)

# Combinar las clases
df_balanced = pd.concat([df_major, df_minor_oversampled])
```

Sin embargo, hay que tener en cuenta que esta técnica es más simple y por ello:

1. **No genera datos sintéticos**: solo duplica o elimina datos existentes.
2. **Mayor riesgo de overfitting** en **oversampling**: duplicar datos reales puede hacer que el modelo se ajuste demasiado a las muestras repetidas.
3. **Sin manejo de ruido**: técnicas como SMOTE o ADASYN generan muestras más variadas, mientras que resample solo replica o elimina datos.

8.2. Tuning de hiperparámetros

El tuning o ajuste de hiperparámetros es un paso crucial para optimizar el rendimiento de un modelo de ML. Los hiperparámetros son parámetros de configuración que debemos definir previamente. Estos pueden influir significativamente en la capacidad del modelo para generalizar y obtener buenos resultados en datos nuevos.

Los hiperparámetros suelen ser valores configurables en cada modelo de ML, que suelen indicarle al modelo cómo debe realizar el entrenamiento. Por ejemplo:

- **`learning_rate` (tasa de aprendizaje):** controla cuánto se ajustan los pesos del modelo con cada iteración.
- **`max_depth`:** profundidad máxima de un árbol.
- **`n_estimators`:** número de árboles en modelos como Random Forest o XGBoost.

El tuning busca encontrar la configuración óptima de hiperparámetros que maximice el rendimiento del modelo en datos nuevos, mejore las métricas o el tiempo de entrenamiento. Ejemplo: en un Random Forest, un número muy bajo de árboles puede resultar en un modelo inestable, mientras que un número muy alto puede aumentar el tiempo de cómputo sin mejorar el rendimiento.

Estrategias para el Tuning de Hiperparámetros

1. Grid Search: consiste en definir un rango de valores para cada hiperparámetro y probar todas las combinaciones posibles de forma exhaustiva. Es ideal para datasets pequeños o pocos hiperparámetros por su alto coste computacional.

Implementación con GridSearchCV en Scikit-learn:

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()

# Definir el espacio de hiperparámetros
param_grid = {
```

```
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10]
}

# Configurar la búsqueda y ver resultados
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5,
scoring='accuracy')
grid_search.fit(X_train, y_train)
print("Mejores hiperparámetros:", grid_search.best_params_)
```

2. Random Search: prueba combinaciones aleatorias dentro de los rangos definidos para cada hiperparámetro. Es eficiente en problemas con muchos hiperparámetros, pero no explora todas las combinaciones posibles.

Implementación con RandomizedSearchCV en Scikit-learn:

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier

# Espacio de hiperparámetros
param_dist = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10]
}

# Configurar la búsqueda y ver resultados
random_search = RandomizedSearchCV(estimator=model,
param_distributions=param_dist, n_iter=10, cv=5, scoring='accuracy')
random_search.fit(X_train, y_train)
print("Mejores hiperparámetros:", random_search.best_params_)
```

3. Búsqueda Bayesiana: utiliza modelos probabilísticos para encontrar combinaciones óptimas de hiperparámetros de manera iterativa. Reduce el número de evaluaciones necesarias y es más eficiente que los anteriores en problemas complejos.

Ejemplo con Optuna:

```
import optuna
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

# Definir la función objetivo
def objective(trial):
    n_estimators = trial.suggest_int('n_estimators', 50, 300)
    max_depth = trial.suggest_int('max_depth', 3, 15)
    min_samples_split = trial.suggest_int('min_samples_split', 2, 10)

    model = RandomForestClassifier(n_estimators=n_estimators,
max_depth=max_depth, min_samples_split=min_samples_split)
    return cross_val_score(model, X_train, y_train, cv=5,
scoring='accuracy').mean()

# Crear el studio y ver los resultados
```

```
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=50)
print("Mejores hiperparámetros:", study.best_params)
```

8.3. Regularización

La regularización es una estrategia clave en machine learning para mejorar la capacidad de generalización de los modelos y reducir el sobreajuste (overfitting). Este problema ocurre cuando un modelo aprende demasiado bien los datos de entrenamiento, incluyendo ruido y peculiaridades específicas, lo que perjudica su rendimiento en datos nuevos.

La idea principal es añadir un término de penalización al objetivo del modelo (normalmente una función de pérdida), que actúa como un control para que el modelo no se vuelva excesivamente complejo. Los modelos lineales se basan fuertemente en la afirmación anterior, como L1 y L2, sin embargo, los modelos no lineales (árboles, support vector machines, etc…) requieren un ajuste de hiperparámetros para realizar la regularización

1. Regularización en modelos lineales

a) L1 (Lasso)

Este método tiene la ventaja de realizar **selección de características automática**, ya que puede reducir a cero los coeficientes de variables irrelevantes. Es útil cuando tenemos conjuntos de datos de alta dimensión con muchas características y/o cuando se espera escasez de características, es decir, cuando se considera que sólo unas pocas características van a ser relevantes; sin embargo, L1 podría ser poco efectivo si se supone que todas las características son importantes y/o si el conjunto de datos tiene problemas de multicolinealidad

Ejemplo:

```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.1)  # alpha controla la fuerza de la regularización
lasso.fit(X_train, y_train)
```

b) L2 (Ridge)

Es ideal cuando las características están correlacionadas, pero no realiza selección de características (no puede reducir coeficientes a cero).

Ejemplo:

```
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=1.0)
```

```
ridge.fit(X_train, y_train)
```

c) Elastic Net

Elastic Net combina L1 (Lasso) y L2 (Ridge), proporcionando un equilibrio entre selección de características y estabilización del modelo. Es útil en datos con muchas características correlacionadas pero requiere ajustar dos hiperparámetros (alpha y l1_ratio).

Ejemplo:

```
from sklearn.linear_model import ElasticNet

elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X_train, y_train)
```

2. Regularización en modelos basados en árboles

Los modelos como **Árboles de decisión**, **Random Forest** y **Gradient Boosting**, tienen diferentes formas de regularización integradas para evitar el sobreajuste:

a) Restricción de profundidad del árbol: limitar la profundidad máxima de los árboles (max_depth) controla la complejidad del modelo, previniendo que los árboles se ajusten demasiado a los datos de entrenamiento.

Ejemplo:

```
tree = DecisionTreeRegressor(max_depth=5)
```

b) Número mínimo de muestras para dividir un nodo: requiere un número mínimo de muestras (min_samples_split) para dividir un nodo, lo que reduce la probabilidad de crear divisiones específicas para puntos aislados en los datos.

Ejemplo:

```
rf = RandomForestClassifier(min_samples_split=10)
```

c) Subsampling: entrenar cada árbol con una fracción de las muestras (subsample) o características (max_features) reduce la varianza del modelo, mejorando su capacidad de generalización.

Ejemplo en XGBoost:

```
xgb_model = xgb.XGBClassifier(subsample=0.8, colsample_bytree=0.8)
```

3. Regularización en máquinas de soporte vectorial (SVM)

En los modelos SVM la regularización se realiza a través del parámetro **C**, que controla el margen de separación entre las clases.

- **C alto**: Reduce la regularización, permitiendo márgenes más estrechos y mayor precisión en el conjunto de entrenamiento, pero aumenta el riesgo de sobreajuste.
- **C bajo**: Aumenta la regularización, permitiendo márgenes más amplios y una mejor generalización.

Ejemplo:

```
svc = SVC(C=0.1)  # Regularización fuerte
```

4. Detección de la necesidad de regularización

- **Curvas de aprendizaje**: Si el modelo tiene un bajo error en entrenamiento, pero un alto error en validación, es muy probable que esté sobreajustado.
- **Evaluación cruzada**: Compara el rendimiento del modelo en diferentes subconjuntos de datos.
- **Alta dimensionalidad**: Si hay muchas características y pocas muestras, la regularización es crucial.

La regularización es una herramienta esencial para mejorar el rendimiento de los modelos de machine learning tradicionales. Desde penalizaciones como L1, L2 y Elastic Net hasta configuraciones específicas en modelos basados en árboles y máquinas de soporte vectorial, estas técnicas ayudan a encontrar el equilibrio perfecto entre ajuste y generalización. La clave está en entender el problema y los datos para seleccionar la técnica adecuada.

8.4. Validación cruzada

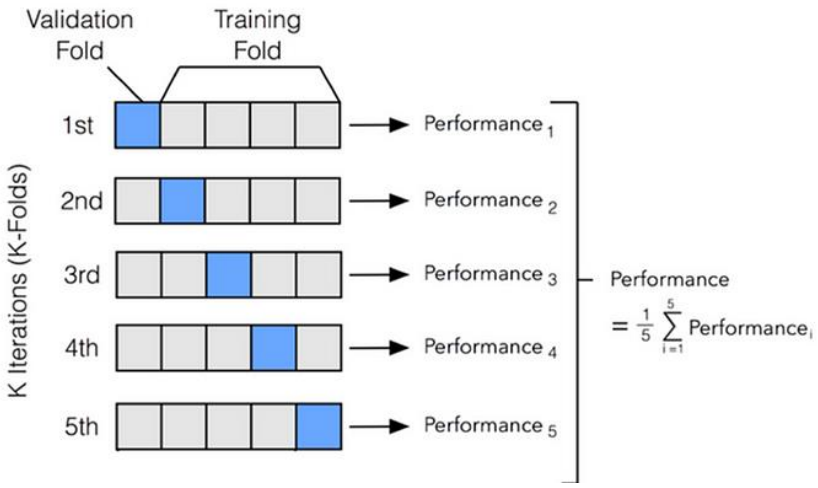
La validación cruzada (Cross-Validation) es una técnica utilizada en ML para evaluar el rendimiento de un modelo y evitar problemas como el sobreajuste (overfitting) o subajuste (underfitting). Consiste en dividir el conjunto de datos en múltiples particiones o "folds" y entrenar y evaluar el modelo varias veces, utilizando cada fold como conjunto de prueba una vez. Existen varios métodos de validación cruzada que se utilizan según el problema y el tipo de datos:

1. K-Fold Cross-Validation

Es el método más común. El conjunto de datos se divide en *K* partes (o "folds") de tamaño similar, y se entrena *K* veces, cada vez utilizando *K-1* folds como entrenamiento y el fold restante como prueba. Al final, las métricas se promedian para obtener el rendimiento final.

2. Stratified K-Fold Cross-Validation

Es una variación del K-Fold. Se asegura de que cada fold tenga una distribución similar de las clases, lo cual es importante en problemas **desbalanceados**. Es útil en clasificación.



Veamos un ejemplo de implementación de KFold:

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

# Modelo
model = RandomForestClassifier()

# Validación cruzada
scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')

print("Scores por fold:", scores)
print("Media de accuracy:", scores.mean())
```

8.5. Ensemble learning (modelos ensamblados)

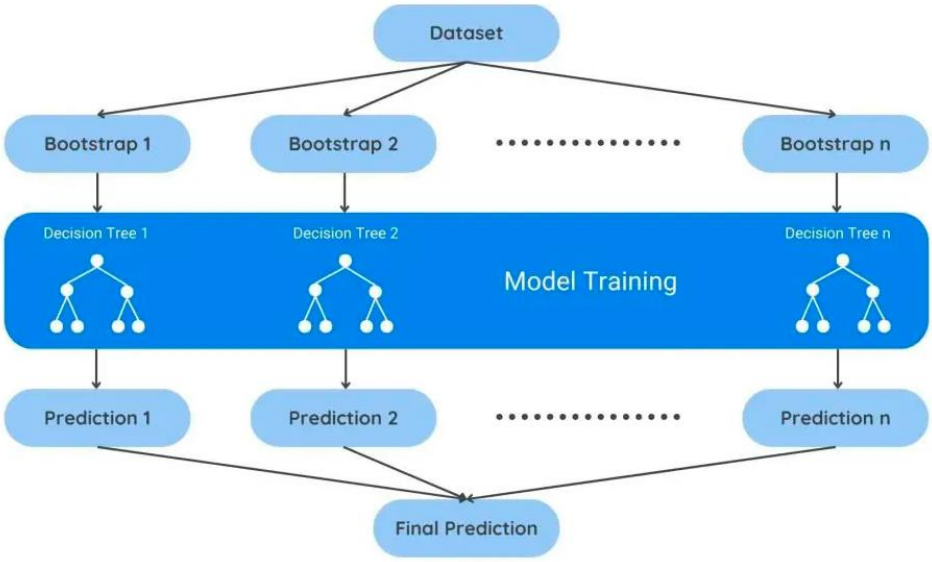
Los modelos ensamblados son técnicas avanzadas en ML que combinan múltiples modelos base para mejorar la precisión y robustez del modelo final. La idea principal es que la combinación de varios modelos, aunque individualmente no sean perfectos, puede dar lugar a un modelo mucho más sólido que los modelos individuales.

Técnicas principales

1. Bagging (Bootstrap Aggregating)

Bagging crea múltiples subconjuntos de los datos de entrenamiento usando muestreo con reemplazo (bootstrap sampling). Cada subconjunto entrena un modelo base, y la predicción final se obtiene promediando (para regresión) o mediante votación (para clasificación) de los resultados de todos los modelos.

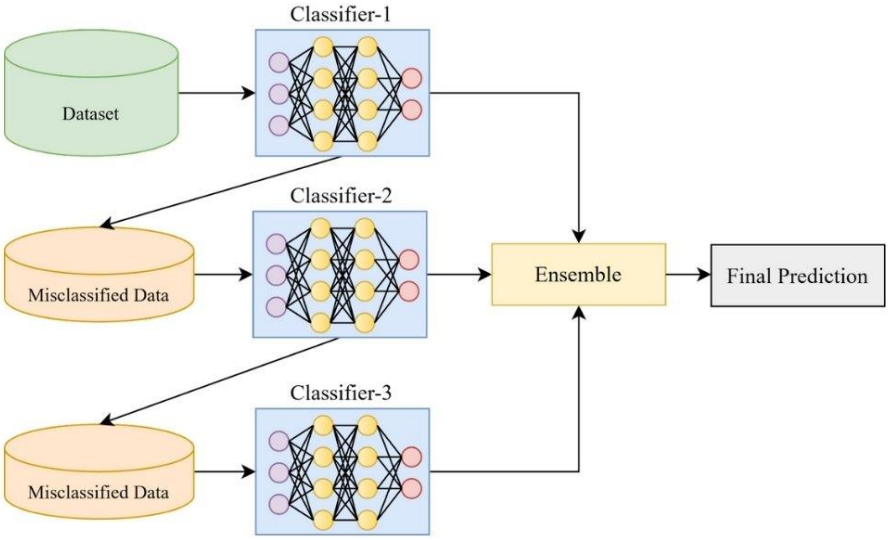
Ejemplo: Random Forest Random Forest es un ejemplo clásico de bagging donde se construyen múltiples árboles de decisión independientes, y el resultado final es la votación mayoritaria o el promedio de las predicciones de todos los árboles.



2. Boosting: combina modelos secuencialmente. Cada modelo se entrena tratando de corregir los errores cometidos por el modelo anterior. Al final, se combinan los modelos con pesos que dependen de su precisión.

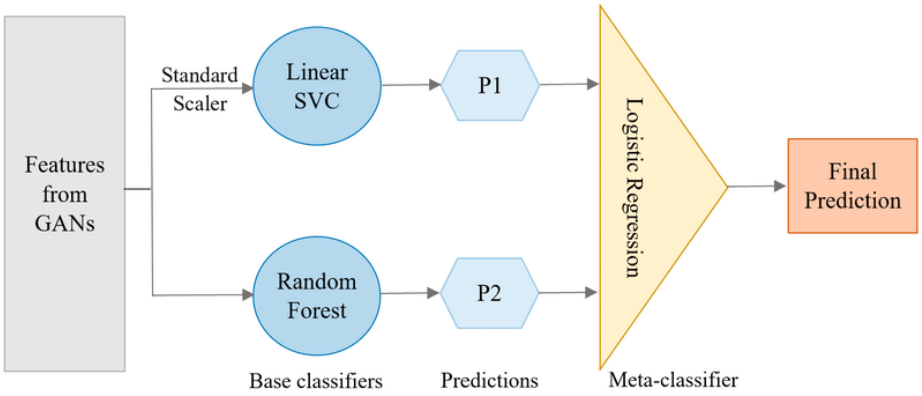
Ejemplos:

a) **AdaBoost** (Adaptive Boosting): entrena modelos débiles secuencialmente y asigna pesos más altos a los errores en cada iteración.
b) **Gradient Boosting**: entrena los modelos secuenciales minimizando la función de pérdida en cada paso.
c) **XGBoost**, **LightGBM** y **CatBoost**: implementaciones optimizadas de boosting que permiten entrenar modelos de manera más eficiente y rápida.



3. Stacking (Stacked Generalization)

Stacking combina diferentes tipos de modelos base (no necesariamente del mismo tipo) y utiliza un meta-modelo para aprender cómo combinar las predicciones de los modelos base. Permite aprovechar lo mejor de varios tipos de algoritmos, y es más flexible y potente que bagging y boosting; pero como contrapartida, tiene mayor complejidad y, por tanto, el tiempo de entrenamiento es más largo.




```
from sklearn.ensemble import RandomForestClassifier, StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

# Modelos base
basemodels = [
    ('svc', SVC(kernel='linear', probability=True, random_state=42)),
    ('rf', RandomForestClassifier(n_estimators=100, random_state=42))
]

# Meta-modelo: regresión logística
model = StackingClassifier(estimators=basemodels,
final_estimator=LogisticRegression())
model.fit(X_train, y_train)
```

4. Voting Classifier: en **voting**, se combinan las predicciones de varios modelos para hacer una predicción final. Hay dos tipos de votación:

- **Votación dura (Hard Voting):** La predicción final se basa en la mayoría de votos.
- **Votación suave (Soft Voting):** Se promedian las probabilidades de predicción.

```
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

# Modelos individuales
```

```
model1 = LogisticRegression()
model2 = SVC(probability=True)
model3 = DecisionTreeClassifier()

# Voting Classifier
voting_clf = VotingClassifier(estimators=[
   ('lr', model1), ('svc', model2), ('dt', model3)
], voting='soft')

voting_clf.fit(X_train, y_train)
```

9. Modelo de clasificación

Basándonos en el mismo dataset con el que estamos trabajando, vamos ahora a abordar la creación de un modelo de clasificación, otro de los tipos de algoritmos supervisados más relevantes. Cambiaremos nuestro objetivo por otro, en esta ocasión queremos poder predecir la existencia de enfermedad cardíaca en base al resto de los atributos del dataset.

Por tanto, ahora, nuestra variable objetivo o "target" será: "cardio". Aprovecharemos para ver otras formas de abordar algunos procedimientos vistos en el desarrollo del ejemplo de regresión.

9.1. Codificación y estandarización con sklearn

La codificación de "gender" ahora la haremos mediante un LabelEncoder, que es una herramienta del módulo sklearn.preprocessing que se utiliza para convertir etiquetas de datos categóricos en números enteros, y lo hace convirtiendo cada categoría única de una variable en un número entero secuencial. Hay que tener en cuenta que es ideal para etiquetas de salida (target) y no tanto para variables independientes (features), para las cuales es mejor usar OneHotEncoder. Aún así vamos a probar con LabelEncoder y dejamos como propuesta contrastar los resultados codificando con OneHotEncoder.

Veamos cómo se haría:

```
le = LabelEncoder()
datos['gender'] = le.fit_transform(datos['gender'])
```

El escalado y normalización, en esta ocasión, lo haremos con un StandardScaler. Se trata de una clase de scikit-learn que se utiliza para escalar características (features) en un conjunto de datos a una distribución con media 0 y desviación estándar 1. Es una de las técnicas más comunes para la normalización de datos en Machine Learning, ya que muchas veces los algoritmos funcionan mejor cuando las características están estandarizadas.

StandardScaler transforma cada característica X (columna de datos) según la siguiente fórmula:

$$Z = \frac{X - \mu}{\sigma}$$

Donde:

- Z: valor estandarizado (output).
- X: valor original de la característica.
- μ: media de la característica (calculada en el conjunto de entrenamiento).

- σ: desviación estándar de la característica (calculada en el conjunto de entrenamiento).

Veamos cómo se haría (reflexiona: ¿se te ocurre alguna forma de optimizar este bloque de código?):

```
se = StandardScaler()
datos['age'] = se.fit_transform(datos[['age']])
datos['weight'] = se.fit_transform(datos[['weight']])
datos['height'] = se.fit_transform(datos[['height']])
datos['ap_hi'] = se.fit_transform(datos[['ap_hi']])
datos['ap_lo'] = se.fit_transform(datos[['ap_lo']])
```

Recuerda: no es necesario escalar ni normalizar los datos cuando usas algoritmos basados en árboles de decisión y sus derivados, como Random Forest, Gradient Boosting (e.g., XGBoost, LightGBM) o AdaBoost.

9.2. Preparación de los conjuntos train y test

Una vez hemos realizado la conversión de categorías a variables numéricas y hemos estandarizado, podemos continuar preparando los conjuntos de entrenamiento y validación previos al entrenamiento de los diferentes algoritmos.

Prepararemos nuevos dataframes con las variables de decisión (X) y el de la variable target (y), así como los conjuntos de entrenamiento y validación:

```
X = datos.drop('cardio',axis=1)
y = datos['cardio']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

9.3. Entrenamiento y validación de modelos de clasificación

Dado que ahora hemos pasado a un modelo de clasificación, las métricas a usar son diferentes al caso de regresión. Para trabajar más cómodamente vamos a crear una función llamada “train_validation” que se va a encargar de:

1. Entrenar el modelo con los conjuntos de train.
2. Probar el modelo con el conjunto de validación.
3. Mostrar las métricas así como la matriz de confusión para valorar la calidad del modelo.

Veamos como quedaría nuestra función:

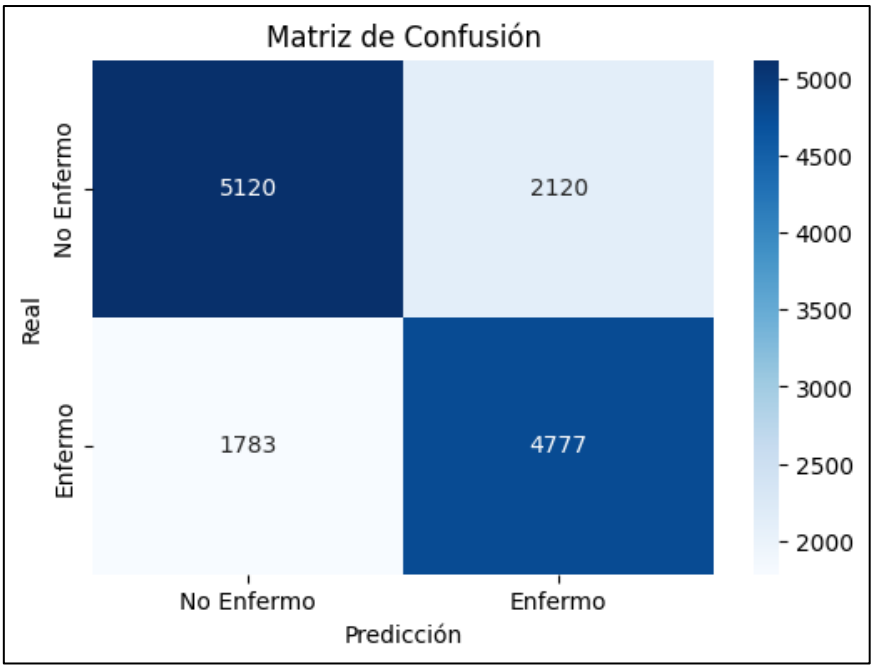
```
def train_validation(model):
    model.fit(X_train,y_train.astype(int))
    y_pred = model.predict(X_test)
    conf_matrix = confusion_matrix(y_pred, y_test)
    print(classification_report(y_pred, y_test.astype(int)))
    print('score_test = ', model.score(X_test, y_test.astype(int)))
    print('score_train = ', model.score(X_train, y_train.astype(int)))

    plt.figure(figsize=(6, 4))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=["No Enfermo", "Enfermo"], yticklabels=["No Enfermo",
"Enfermo"])
    plt.title("Matriz de Confusión")
    plt.xlabel("Predicción")
    plt.ylabel("Real")
    plt.show()
```

Si probamos con un primer modelo de clasificación, por ejemplo, un RandomForestClassifier:

```
model = RandomForestClassifier(random_state=42, n_estimators=100)
train_validation(model)
```

Podemos obtener los siguientes resultados:

```
              precision    recall  f1-score   support

           0       0.74      0.71      0.72      7240
           1       0.69      0.73      0.71      6560

    accuracy                           0.72     13800
   macro avg       0.72      0.72      0.72     13800
weighted avg       0.72      0.72      0.72     13800

score_test =  0.7171739130434782
score_train =  1.0
```



Reflexiona: ¿qué podemos deducir de las métricas obtenidas?, ¿crees que es un buen modelo?

10. Métricas para validación de la clasificación

En el apartado 6 vimos métricas para validar un modelo de regresión; dado que en el ejemplo anterior se han introducido ya algunas para clasificación, es momento de formalizarlas.

Partiremos de la matriz de confusión (confusion matrix, CM), la cual presenta en una tabla una visión gráfica de los errores cometidos por el modelo de clasificación. Se trata de un modelo gráfico para visualizar el nivel de acierto de un modelo de predicción.

Es habitual clasificar entre 2 categorías, por lo que la matriz de confusión tendrá esta forma:

		Clase predicha	
		P	N
Clase verdadera	P	TP	FN
	N	FP	TN

Siendo:

- **Verdadero positivo** (true positive, TP): número de clasificaciones correctas en la clase positiva (P).
- **Verdadero negativo** (true negative, TN): número de clasificaciones correctas en la clase negativa (N).
- **Falso negativo** (false negative, FN): número de clasificaciones incorrectas de clase positiva clasificada como negativa.
- **Falso positivo** (false positive, FP): número de clasificaciones incorrectas de clase negativa clasificada como positiva.

La matriz, además, nos proporciona la posibilidad de calcular determinadas métricas con los valores existentes en ella. Concretamente nos interesan las siguientes:

- **Exactitud** (accuracy): es el número de predicciones correctas sobre el número total de predicciones. Es una métrica global que no distingue entre clases, fácil de interpretar, pero no adecuada para datasets desbalanceados.

$$ACC = \frac{TP + TN}{FP + FN + TP + TN}$$

- **Precisión**: mide la proporción de predicciones positivas correctas entre todas las predicciones positivas realizadas por el modelo. Es útil cuando el coste de un falso positivo es alto.

$$PRE = \frac{TP}{TP + FP}$$

- **Sensibilidad** (recall): mide la capacidad del modelo para identificar correctamente todas las instancias positivas. Útil cuando los falsos negativos son graves (por ejemplo, en diagnóstico médico).

$$REC = SEN = TPR = \frac{TP}{FN + TP}$$

- **Especificidad** (specificity): tasa de instancias correctamente clasificadas como negativas respecto a todas las instancias negativas.

$$SPE = \frac{TN}{TN + FP} = 1 - FPR$$

- **F1-Score**: métrica que combina precisión y sensibilidad (recall) en un solo valor. Útil cuando las clases están desequilibradas y se necesita un compromiso entre precisión y exhaustividad. Si el F1-score es bajo, el modelo necesita mejorar tanto en precisión como en exhaustividad.

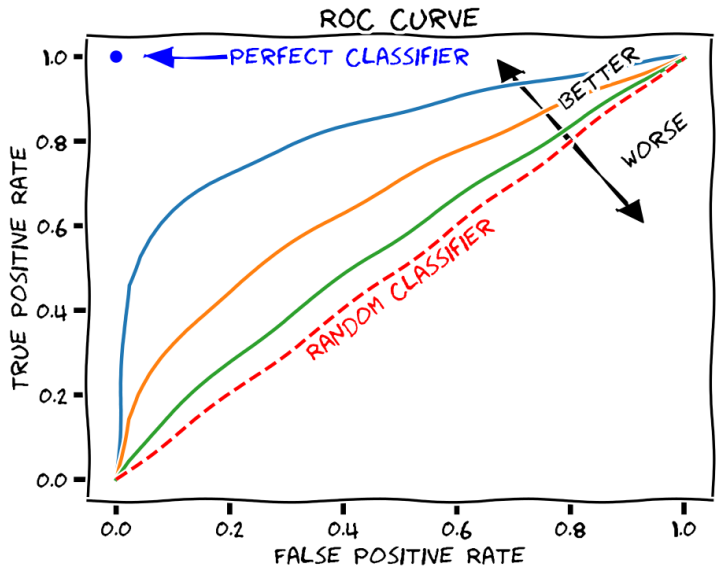
$$F1 = 2 \times \frac{PRE \times REC}{PRE + REC}$$

Además de las métricas anteriores, también es importante la **ROC-AUC** (Curva ROC y área bajo la curva). La AUC mide la capacidad del modelo para separar correctamente las clases positivas y negativas. Se basa en la curva ROC, que representa la relación entre la tasa de verdaderos positivos (exhaustividad) y la tasa de falsos positivos.

Interpretación:

- AUC cerca de 1: excelente separación entre clases.
- AUC cerca de 0.5: separación aleatoria (modelo no útil).
- AUC bajo: el modelo clasifica incorrectamente, con resultados peores que el propio azar.

En la siguiente imagen podemos apreciar gráficamente el funcionamiento de una curva ROC. El clasificador ideal sería el que situaría su “curva” lo más hacia la esquina superior izquierda posible. Por esa razón, el modelo de la curva azul es mejor que el de la curva naranja, y éste es mejor que el modelo de la curva verde.



Una ventaja que presentan las curvas ROC es que nos ayudan a encontrar un umbral de clasificación que se adapte a nuestro problema específico.

Por ejemplo, si estuviéramos evaluando un clasificador de correo no deseado, querríamos que la tasa de falsos positivos fuera realmente baja. No queríamos que nadie perdiera un correo electrónico importante en el filtro de spam solo porque nuestro algoritmo es demasiado agresivo. Probablemente, incluso permitiríamos una buena cantidad de correos electrónicos no deseados reales (verdaderos positivos) a través del filtro solo para asegurarnos de que no se perdieran correos electrónicos importantes.

Por otro lado, si nuestro clasificador estuviera prediciendo si alguien tiene una enfermedad terminal, podríamos estar de acuerdo con un mayor número de falsos positivos (diagnósticos incorrectos de la enfermedad), solo para asegurarnos de que no perdamos ningún positivo verdadero (personas que realmente tienen la enfermedad).

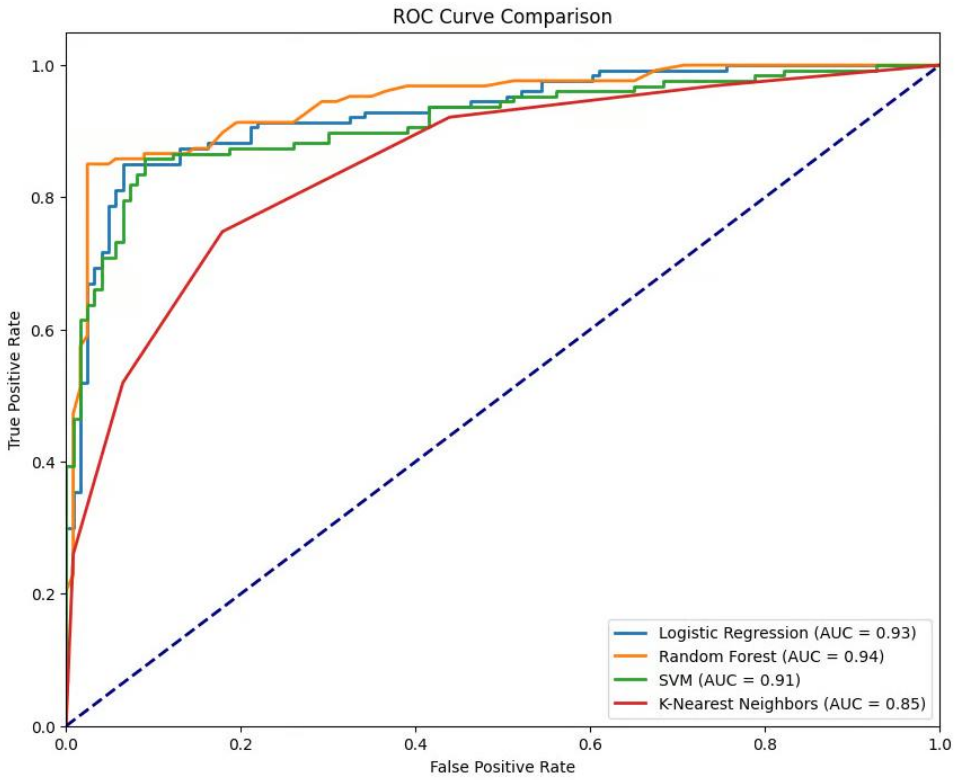
Además, las curvas ROC y las puntuaciones AUC también nos permiten comparar el rendimiento de diferentes clasificadores para el mismo problema y así poder elegir el de mayor rendimiento.

Veamos un ejemplo para comparar el rendimiento de 4 algoritmos de clasificación usando ROC.

```
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score,
recall_score, f1_score, confusion_matrix, roc_curve, auc
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

# Generate a binary classification dataset
```

```
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2,
random_state=0)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)

# Define the models
models = {
    "Logistic Regression": LogisticRegression(),
    "Random Forest": RandomForestClassifier(),
    "SVM": SVC(probability=True),
    "K-Nearest Neighbors": KNeighborsClassifier()
}

# Initialize a dictionary to store AUC - ROC scores
roc_auc_scores = {}

# Plot the ROC curves
plt.figure(figsize=(10, 8))

for name, model in models.items():
    # Train the model
    model.fit(X_train, y_train)

    # Predict the probabilities
    y_probs = model.predict_proba(X_test)[:, 1]

    # Calculate the AUC - ROC score
    roc_auc = roc_auc_score(y_test, y_probs)
    roc_auc_scores[name] = roc_auc

    # Compute ROC curve
    fpr, tpr, _ = roc_curve(y_test, y_probs)

    # Plot ROC curve
    plt.plot(fpr, tpr, lw=2, label=f'{name} (AUC = {roc_auc:.2f})')

# Plot the diagonal 50% line
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

# Customize the plot
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend(loc="lower right")
plt.show()

# Print the AUC - ROC scores for each model
for name, score in roc_auc_scores.items():
    print(f'{name}: AUC - ROC = {score:.2f}')
```

Veamos la gráfica resultante:



La figura anterior muestra las curvas correspondientes a los 4 clasificadores, donde el RandomForestClassifier es el que mejor funciona, con un AUC-ROC de 0,94, mientras que el KNeighborsClassifier es el que “peor” funciona, con un AUC-ROC de 0,85. Un modelo ideal abrazaría la esquina superior izquierda del gráfico, donde se maximiza el TPR y se minimiza el FPR. Cuanto más se acerque la curva a este punto, mejor será el rendimiento del modelo.