Universidad Carlos III

Facultad de Ciencias Sociales y Jurídicas
Estadística y Empresa
Aprendizaje Automático

Clasificación de pacientes epáticos mediante KNN y Árboles de clasificación

Práctica 1

Noa Castro González y Jesús Secilla Martínez

Grupo E

Noviembre 2023

Índice

1. Introducción

La aplicación de algoritmos de aprendizaje automático en la resolución de problemas y la toma de decisiones se ha convertido en un componente esencial en diversas áreas de la ciencia y la industria. En este trabajo se utilizará un conjunto de datos denominado *ILPD*, que contiene información sobre pacientes con enfermedades de hígado. El objetivo principal es extraer información de este conjunto de datos y construir modelos predictivos sólidos utilizando dos enfoques distintos: el algoritmo k-Nearest Neighbors (k-NN) y un árbol de clasificación.

Se comenzará realizando un Análisis Exploratorio de Datos (EDA) que nos permitirá comprender en profundidad la naturaleza del conjunto de datos. Durante esta fase, se explorará el volumen y el tipo de datos con los que se trabaja, se identificará si existen valores faltantes y se evaluará la necesidad de normalizar las variables. Este paso crucial sentará las bases para el posterior análisis y construcción de los algoritmos.

A continuación, se comenzará aplicando el algoritmo k-NN, donde se llevará a cabo un preprocesamiento de los datos para posteriormente ajustar los hiperparámetros utilizando k-fold validation (con k=10). El objetivo será maximizar la precisión del modelo y comprender su rendimiento en la clasificación. Luego, se entrenará el modelo con los mejores hiperparámetros y se evaluará su desempeño utilizando una partición de holdout al 20 %, además de una métrica adicional acorde a un problema de clasificación médico para obtener una visión más completa.

Además, se construirá un modelo de árbol de clasificación. En este caso, se empleará una pipeline para el preprocesamiento de datos y se ajustarán los hiperparámetros utilizando un grid search. De nuevo, se evaluará el rendimiento del modelo utilizando k-fold validation (k=10) y una partición de holdout al 20 %, nuevamente junto con una métrica adicional para obtener una comprensión más profunda de su capacidad predictiva.

Este trabajo busca proporcionar una visión integral de la aplicación de técnicas de aprendizaje automático en la resolución de problemas, desde la preparación de datos hasta la construcción y evaluación de modelos.

El código en Python usado a lo largo del trabajo y el conjunto de datos se puede encontrar en el siguiente *enlace*.

2. EDA

En primer lugar, se realizará un breve Análisis Exploratorio de los Datos (EDA). Esta es una etapa muy importante en el proceso de análisis de datos, ya que se exploran y se comprenden los datos con los que se trabaja. Es fundamental obtener información previa sobre los mismos, especialmente para conocer el volumen y tipo de datos que se manipularán. En este caso, se muestra especial interés por saber si los datos necesitan normalización, ya que será necesario para poder aplicar el modelo K-NN ; si existen datos faltantes, para realizar la imputación de los mismos con alguna técnica univariante y una breve visualización de la distribución de las variables para entender mejor el conjunto de datos con el que se trabajará.

```
# Importar librerías
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.neighbors import KNeighborsRegressor, KNeighborsClassifier
from sklearn.metrics import r2_score, accuracy_score, confusion_matrix,
    classification_report
from sklearn.impute import SimpleImputer
from sklearn.model_selection import GridSearchCV, cross_val_score
```

En primer lugar, se procederá a cargar el conjunto de datos con el que se realizará esta práctica, y se almacenan en un DataFrame de pandas.

```
datos = pd.read_csv('ILPD.csv')
df = pd.DataFrame(datos)
df
```

index	Age	Gender	Total_Bilirubin	Direct_Bilirubin	Alkaline_Phosphotase	Alamine_Aminotransferase
0	65	Female	0.7	0.1	187	16
1	62	Male	10.9	5.5	699	64
2	62	Male	7.3	4.1	490	60
3	58	Male	1.0	0.4	182	14
4	72	Male	3.9	2.0	195	27
⋮	⋮	⋮	⋮	⋮	⋮	⋮
577	32	Male	12.7	8.4	190	28
578	60	Male	0.5	0.1	500	20
579	40	Male	0.6	0.1	98	35
580	52	Male	0.8	0.2	245	48
581	31	Male	1.3	0.5	184	29
582	38	Male	1.0	0.3	216	21

Tabla 1: Datos del conjunto de datos (Parte 1)

Por tanto, se comenzará a realizar una breve exploración del conjunto *ILPD*. Se imprirá la forma del DataFrame (número de filas y columnas) para tener una idea de la estructura de los datos que se maneja.

index	Aspartate_Aminotransferase	Total_Protiens	Albumin	Albumin_and_Globulin_Ratio	Dataset
0	18	6.8	3.3	0.9	1
1	100	7.5	3.2	0.74	1
2	68	7.0	3.3	0.89	1
3	20	6.8	3.4	1.0	1
4	59	7.3	2.4	0.4	1
⋮	⋮	⋮	⋮	⋮	⋮
577	47	5.4	2.6	0.9	1
578	34	5.9	1.6	0.37	2
579	31	6.0	3.2	1.1	1
580	49	6.4	3.2	1.0	1
581	32	6.8	3.4	1.0	1
582	24	7.3	4.4	1.5	2

Tabla 2: Datos del conjunto de datos (Parte 2)

Es importante para comprender la dimensión del conjunto de datos.
Se imprimirá una lista con las columnas presentes en el DataFrame, visualizado anteriormente. Esto proporciona una visión general de las variables disponibles.
También se muestran los tipos de datos de cada columna. Esto es fundamental para entender si las columnas se interpretan correctamente como números, texto, fechas u otros tipos. Además, esto indicará a qué variables será necesario aplicarle alguna transformación.
Por último, se calculan estadísticas descriptivas para las variables cuantitativas. Esto incluye información sobre la tendencia central, la dispersión y la distribución de los datos.

```
# Explorar el dataframe
print(df.shape)
print(df.columns)
print(df.dtypes)
print(df.iloc[:, :-1].describe())
```

```
(583, 11)
Index(['Age', 'Gender', 'Total_Bilirubin', 'Direct_Bilirubin',
       'Alkaline_Phosphotase', 'Alamine_Aminotransferase',
       'Aspartate_Aminotransferase', 'Total_Protiens', 'Albumin',
       'Albumin_and_Globulin_Ratio', 'Dataset'],
      dtype='object')
Age                           int64
Gender                       object
Total_Bilirubin             float64
Direct_Bilirubin            float64
Alkaline_Phosphotase          int64
Alamine_Aminotransferase      int64
Aspartate_Aminotransferase    int64
Total_Protiens              float64
Albumin                     float64
Albumin_and_Globulin_Ratio  float64
Dataset                       int64
dtype: object
              Age  Total_Bilirubin  Direct_Bilirubin  Alkaline_Phosphotase  \
```

	Age	Total_Bilirubin	Direct_Bilirubin	Alkaline_Phosphotase
count	583.000000	583.000000	583.000000	583.000000
mean	44.746141	3.298799	1.486106	290.576329
std	16.189833	6.209522	2.808498	242.937989
min	4.000000	0.400000	0.100000	63.000000
25%	33.000000	0.800000	0.200000	175.500000
50%	45.000000	1.000000	0.300000	208.000000
75%	58.000000	2.600000	1.300000	298.000000
max	90.000000	75.000000	19.700000	2110.000000

	Alamine_Aminotransferase	Aspartate_Aminotransferase	Total_Protiens	\
count	583.000000	583.000000	583.000000	
mean	80.713551	109.910806	6.483190	
std	182.620356	288.918529	1.085451	
min	10.000000	10.000000	2.700000	
25%	23.000000	25.000000	5.800000	
50%	35.000000	42.000000	6.600000	
75%	60.500000	87.000000	7.200000	
max	2000.000000	4929.000000	9.600000	

	Albumin	Albumin_and_Globulin_Ratio
count	583.000000	579.000000
mean	3.141852	0.947064
std	0.795519	0.319592
min	0.900000	0.300000
25%	2.600000	0.700000
50%	3.100000	0.930000
75%	3.800000	1.100000
max	5.500000	2.800000

Tras esta exploración del DataFrame, se puede ver como todas las variables son númericas excepto *Gender*, que es tipo `object`, pues contiene el género del paciente. Además, la variable respuesta, que es *Dataset* se ha eliminado para realizar las estadísticas descriptivas, ya que es una variable categórica que viene expresada en números enteros siendo **1: pacientes hepáticos** y **2: pacientes sanos**.
También cabe resaltar las escalas de cada variable. Se observa como cada una presenta una media, cuartiles, desviación y demás estadísticas muy diferentes entre sí. Esto hace pensar que las variables necesitarán normalización para los algoritmos que lo requieran, en este caso el k-NN, ya que se basa en la distancia entre puntos y la escala de las variables puede influir en el resultado.

Como se puede apreciar, se ve que las dimensiones del DataFrame son **583 x 11**. Todas las variables presentan 583 registros salvo la última, `Albumin and Globulin Ratio`, que presenta 579 observaciones, es decir, 4 instancias sin datos. Para comprobarlo, se ejecutará el siguiente código, en busca de valores faltantes:

```
# Calcular el número de valores faltantes por columna
missing_values_count = df.isnull().sum()

# Calcular el porcentaje de valores faltantes por columna
missing_values_percent = 100 * missing_values_count / len(df)

# Crear un dataframe con el número y el porcentaje de valores faltantes
missing_values_df = pd.DataFrame({'count': missing_values_count, 'percent':
    missing_values_percent})

# Mostrar el dataframe ordenado por porcentaje de forma descendente
missing_values_df.sort_values('percent', ascending=False)
```

Index	Count	Percent
Albumin_and_Globulin_Ratio	4	0.6861063464837049
Age	0	0.0
Gender	0	0.0
Total_Bilirubin	0	0.0
Direct_Bilirubin	0	0.0
Alkaline_Phosphotase	0	0.0
Alamine_Aminotransferase	0	0.0
Aspartate_Aminotransferase	0	0.0
Total_Protiens	0	0.0
Albumin	0	0.0
Dataset	0	0.0

Se observa como se cumple la hipótesis planteada, `Albumin and Globulin Ratio` es la única variable que presenta instancias vacías, un total de 4. Esto representa el 0.68 % del conjunto de datos total.

También se visualizarán histogramas y diagramas de dispersión para comprobar la diferente escala de las variables.

```
# Generar histogramas para las variables numéricas
df.iloc[:, :-1].hist(bins=20, figsize=(15, 10))
plt.show()
```

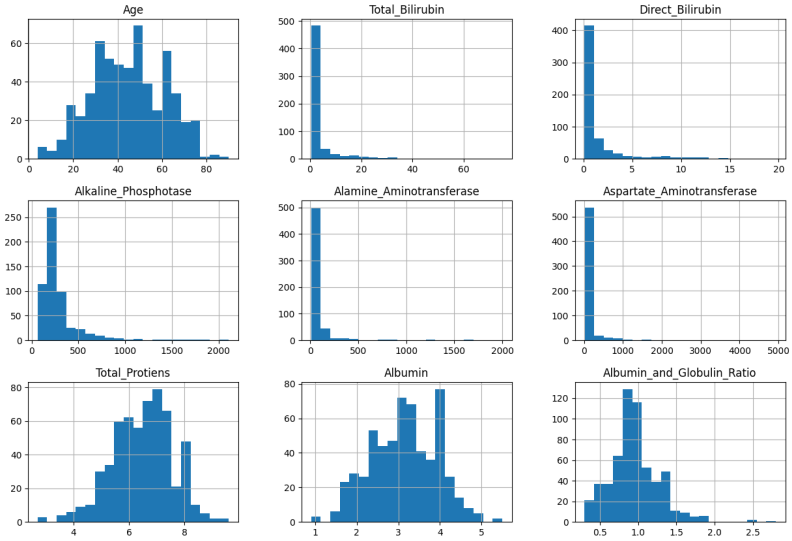

Figura 1: Histogramas de las variables cuantitativas

```
# Generar una matriz de diagramas de dispersión para las variables numéricas
pd.plotting.scatter_matrix(df.iloc[:, :-1], figsize=(15,10), diagonal='kde')
plt.show()
```

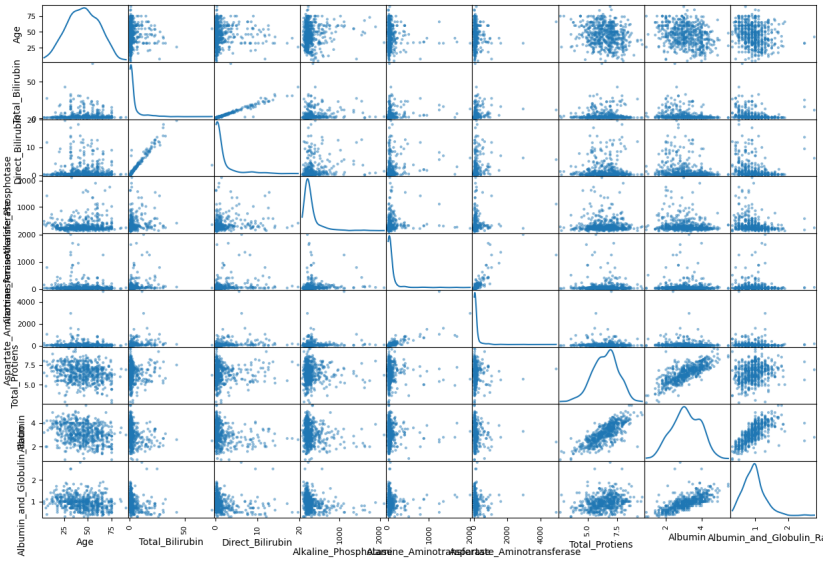

Figura 2: Diagrama de dispersión de las variables cuantitativas

En los histogramas se puede ver cómo claramente cada variable se concentra en torno a unos valores, por lo que tienen escalas distintas. Al graficar las variables en el diagrama de dispersión, se puede notar lo mismo; cómo las variables se distribuyen en rangos de valores distintos. Se puede ver como para cada par de variables se tienen escalas diferentes.

Por tanto, se concluye tras este breve Análisis Exploratorio que existen datos faltantes para la variable `Albumin and Globulin Ratio`, que posteriormente en los preprocesados de cada algoritmo se imputarán con alguna métrica univariante (media, mediana, moda...). Además, también se ha observado las diferentes escalas y distribución de los datos para cada variable, por lo que será necesario normalización para aplicar el algoritmo k-NN.

3. Construcción de un modelo KNN

3.1. ¿Qué es y como funciona?

El k-NN es un algoritmo de aprendizaje perezoso (*lazy learning*) porque no entrena un modelo con los datos de entrenamiento en el sentido tradicional. En cambio, realiza predicciones basadas en la similitud entre los puntos de datos, es decir, basándose en sus vecinos más cercanos en un conjunto de datos existente. La idea central detrás del algoritmo k-NN es que los puntos de datos similares tienden a pertenecer a la misma clase o tener valores de salida similares. Funciona de la siguiente manera:

Se tiene un conjunto de datos en el que cada punto representa un objeto o una entidad con ciertas características. El k-NN clasifica o predice un nuevo punto asignándole la etiqueta de clase más común entre sus K vecinos más cercanos en ese conjunto de datos. La elección de K se realiza antes de aplicar el algoritmo y afecta la sensibilidad del modelo a los datos.

Para hacer esto, el k-NN calcula la distancia entre el nuevo punto y todos los puntos existentes en el conjunto de datos, utilizando una métrica de distancia, un hiperparámetro que claramente se puede ajustar. Los puntos más cercanos al nuevo punto se denominan sus "vecinos más cercanos".

Si K es igual a 1, se toma la etiqueta de clase del punto más cercano como la predicción. Si K es mayor que 1, se realiza una especie de "votación"entre los K vecinos más cercanos, y la etiqueta de clase más común se asigna al nuevo punto como su predicción.

En resumen, el k-NN decide la etiqueta de clase o el valor de un nuevo punto basándose en la mayoría de las etiquetas o valores de sus vecinos más cercanos en el conjunto de datos. Es un enfoque simple y efectivo para la clasificación y regresión, especialmente cuando los datos tienen una estructura simple y bien definida.

3.2. Preprocesado de datos

La etapa de preprocesado de datos desempeña un papel fundamental en la construcción de un modelo de aprendizaje automático óptimo. En esta fase, se realizan varias tareas esenciales para asegurar que los datos estén listos para ser utilizados por el modelo. Como se ha mencionado anteriormente, es necesario tratar los datos para que estén en un formato compatible para su análisis y puedan ser interpretados por el algoritmo k-NN de manera efectiva. Por tanto, se realizará la la imputación de valores faltantes,

codificación de variables categóricas mediante one-hot encoding y la normalización de los predictores. También será necesario dividir los datos en conjunto de entrenamiento y prueba.

El primero de los pasos será la división de los datos en predictores (X) y la variable objetivo (y). El conjunto X lo conforman las columnas que se utilizarán para hacer predicciones, mientras que la variable objetivo es la que se intentará predecir, en este caso `Dataset`.

```
# Dividir el dataframe en X e y
X = df.drop("Dataset", axis=1)
y = df["Dataset"]
print(X.info())
print("\n")
print(y.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 583 entries, 0 to 582
Data columns (total 10 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   Age                         583 non-null    int64
 1   Gender                      583 non-null    object
 2   Total_Bilirubin             583 non-null    float64
 3   Direct_Bilirubin            583 non-null    float64
 4   Alkaline_Phosphotase        583 non-null    int64
 5   Alamine_Aminotransferase    583 non-null    int64
 6   Aspartate_Aminotransferase  583 non-null    int64
 7   Total_Protiens              583 non-null    float64
 8   Albumin                     583 non-null    float64
 9   Albumin_and_Globulin_Ratio  579 non-null    float64
dtypes: float64(5), int64(4), object(1)
memory usage: 45.7+ KB
None
```

El siguiente paso será dividir los datos en conjuntos de entrenamiento y prueba. El conjunto de entrenamiento se utilizará para entrenar el modelo, mientras que el conjunto de prueba se utilizará para evaluar su rendimiento. Tal y como se pide, se ha asignado el 80 % de los datos al conjunto de entrenamiento (X_train, y_train) y el 20 % restante al conjunto de prueba (X_test, y_test). Se fija una semilla `random_state=42` para garantizar que la división sea reproducible.

```
# Dividir aleatoriamente el conjunto de datos en entrenamiento y prueba con Holdout 20%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

La razón para dividir los datos en conjuntos de entrenamiento y prueba antes del preprocesamiento de datos se relaciona directamente con la prevención del *data leakage*. El *data leakage* es una situación en la que información no deseada o inapropiada de los datos de prueba se filtra accidentalmente al conjunto de entrenamiento, lo que puede llevar a una evaluación excesivamente optimista del modelo.

El siguiente paso será codificar las variables categóricas. En este conjunto de datos, hay una variable categórica llamada `Gender` que necesita ser codificada para ser utilizada. Se aplicará la codificación one-hot a esta variable utilizando la función **apply_one_hot_encoding**:

```
def apply_one_hot_encoding(df, factor_columns):
    # Crea el codificador OneHotEncoder
    encoder = OneHotEncoder(sparse=False)

    # Selecciona las columnas de factor o strings
    factor_data = df[factor_columns]

    # Aplica la codificación OneHotEncoder a las columnas de factor
    encoded_data = encoder.fit_transform(factor_data)

    # Crea un nuevo dataframe con las columnas codificadas
    encoded_columns = encoder.get_feature_names_out(factor_columns)
    encoded_df = pd.DataFrame(encoded_data, columns=encoded_columns, index=df.index)

    print(encoded_df.head())

    # Une el nuevo data frame con las columnas codificadas al dataframe original
    final_df = pd.concat([df.drop(factor_columns, axis=1), encoded_df], axis=1)

    return final_df
```

La codificación one-hot convierte categorías en columnas binarias (0 o 1), lo que permite su uso en el modelo. Se procede a aplicarla en los conjuntos de X, entrenamiento y test, señalando antes cuáles son las columnas de tipo factor a convertir en binarias.

```
#Definir qué variables del dataframe son factores
factor_columns = ['Gender']

#Ejectutar función
xtrain_final = apply_one_hot_encoding(X_train, factor_columns)
xtest_final = apply_one_hot_encoding(X_test, factor_columns)
```

```
     Gender_Female  Gender_Male
77             1.0          0.0
581            0.0          1.0
210            0.0          1.0
192            0.0          1.0
449            1.0          0.0
     Gender_Female  Gender_Male
355            0.0          1.0
407            0.0          1.0
90             0.0          1.0
402            1.0          0.0
268            0.0          1.0
```

Se observa como ahora la variable `Gender` se ha transformado en dos variables dummy que permiten expresar el género del paciente.

El próximo paso será la imputación de datos faltantes. Se ha utilizado **SimpleImputer** con la estrategia 'mean' para imputar valores faltantes en los datos de entrenamiento y prueba. La estrategia 'mean' significa que los valores faltantes se han reemplazado por la media de la columna correspondiente, en este

caso `Albumin and Globulin Ratio`. De esta manera se garantiza que el DataFrame esté completo antes de aplicar el modelo.

```
# Crear un objeto SimpleImputer con la estrategia "mean" (media)
imputer = SimpleImputer(strategy='mean')

# Ajustar y transformar los datos con el imputador en tu conjunto de entrenamiento
xtrain_final = imputer.fit_transform(xtrain_final)

# Transformar los datos de prueba con el mismo imputador
xtest_final = imputer.transform(xtest_final)
```

El último paso del preprocesado será la normalización de las variables. Esta es una práctica común en k-NN, que permite asegurar de que todas las variables contribuyan de manera equitativa a la métrica de distancia. Al estandarizar las variables, ajustamos sus escalas para que tengan una media de 0 y una desviación estándar de 1. Esto garantiza que todas las variables tengan un impacto igual en la distancia euclidiana y, por lo tanto, en la clasificación.

```
# Normalizar los datos
scaler = StandardScaler()
xtrain_final = scaler.fit_transform(xtrain_final)
xtest_final = scaler.transform(xtest_final)
```

La razón por la que se aplica scaler.fit_transform a los datos de entrenamiento y scaler.transform a los datos de prueba en la normalización se basa en la buena práctica de ajustar el escalador (scaler) solo a los datos de entrenamiento y luego aplicar la misma transformación a los datos de prueba. Lo mismo ocurre con la imputación de datos faltantes.
De esta manera, se ha obtenido un conjunto de datos normalizado, sin datos faltantes y sin variables categóricas no codificadas. Teniendo los datos preparados, se puede proceder a la construcción del k-NN.

3.3. Ajuste de hiperparámetros

La configuración adecuada de los hiperparámetros es esencial para la construcción de un modelo de aprendizaje automático óptimo. En esta etapa, se pretende buscar los mejores hiperparámetros que optimicen el rendimiento del modelo. Se comienza por definir una serie de hiperparámetros que se desea ajustar, así como variables para realizar un seguimiento de los mejores hiperparámetros y la mejor precisión encontrada hasta el momento. Se emplearán tres bucles anidados para recorrer todas las combinaciones posibles de hiperparámetros. Cada bucle itera sobre un hiperparámetro diferente, y en conjunto, se exploran todas las combinaciones posibles. La validación cruzada con 10 folds nos proporciona una evaluación sólida de cada configuración, lo que nos permite identificar los hiperparámetros que maximizan la precisión del modelo.

Los hiperparámetros que se ajustarán serán los siguientes:

- **n_neighbors**: Este hiperparámetro representa el número de vecinos más cercanos que se utilizarán para tomar una decisión de clasificación. En otras palabras, k-NN clasifica un punto de datos basándose en la mayoría de votos de sus k vecinos más cercanos.

- **metric**: Este hiperparámetro determina la métrica de distancia que se utilizará para medir la distancia entre puntos y decidir quiénes son los vecinos más cercanos. Entre las opciones de métricas se encuentran la distancia euclídea, la de Manhattan, la de Chebyshev y la de Minkowski.
- **weights**: Este hiperparámetro determina cómo se ponderan los votos de los vecinos en función de su distancia. Puede tomar dos valores:
  - `uniform`: si todos los vecinos tienen el mismo peso en la decisión de clasificación.
  - `distance`: si los votos de los vecinos se ponderan inversamente a su distancia. Vecinos más cercanos tienen más influencia en la votación que vecinos más lejanos.

```
best_accuracy = 0
best_hyperparameters = {}

for n_neighbors in range(3, 31, 2):
    for metric in ['euclidean', 'manhattan', 'chebyshev', 'minkowski']:
        for weights in ['uniform', 'distance']:
            knn = KNeighborsClassifier(n_neighbors=n_neighbors, metric=metric, weights=
    weights)
            scores = cross_val_score(knn, xtrain_final, y_train, cv=10, scoring='accuracy'
    )
            average_accuracy = scores.mean()

            if average_accuracy > best_accuracy:
                best_accuracy = average_accuracy
                best_hyperparameters = {
                    'n_neighbors': n_neighbors,
                    'metric': metric,
                    'weights': weights
                }

# Imprime los mejores hiperparámetros encontrados
print("Mejores hiperparámetros:", best_hyperparameters)
print("Precisión promedio:", best_accuracy)
```

```
Mejores hiperparámetros: {'n_neighbors': 29, 'metric': 'chebyshev', 'weights': 'distance'}
Precisión promedio: 0.7083718778908419
```

Se obtiene que los hiperparámetros que maximizan el rendimiento del modelo con los datos de entrenamiento son `n_neighbors` = 29, `metric` = 'chebyshev' y `weights` = 'distance'.

También se ha comprobado el funcionamiento del código a través de una búsqueda en grid. Se ha definido una cuadrícula de hiperparámetros (`param_grid`) que incluye el número de vecinos (`n_neighbors`), la métrica de distancia (`metric`), y el peso de los vecinos (`weights`). Posteriormente, se realizará una búsqueda en grid que explorará una variedad de combinaciones de los hiperparámetros.

```
# Define los hiperparámetros para la búsqueda en grid
param_grid = {
    'n_neighbors': list(range(1, 31, 2)),
    'metric': ['euclidean', 'manhattan', 'chebyshev', 'minkowski'],
```

```
    'weights': ['uniform', 'distance']
}

# Crea un modelo KNN
knn = KNeighborsClassifier()

# Realiza la búsqueda de hiperparámetros con k-fold validation (k=10)
grid_search = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy', verbose=1)
grid_search.fit(xtrain_final, y_train)

# Imprime los mejores hiperparámetros encontrados
print(f'Mejores hiperparámetros: {grid_search.best_params_}')
print("Precisión promedio:", grid_search.best_score_)
```

```
Fitting 10 folds for each of 120 candidates, totalling 1200 fits
Mejores hiperparámetros: {'metric': 'chebyshev', 'n_neighbors': 29, 'weights': 'distance'}
Precisión promedio: 0.7083718778908419
```

Se observa cómo se obtienen los mismos resultados de mejores hiperparámetros y precisión promedio para ambas maneras de calcularlos. A través del GridSeacrh se realiza una búsqueda más eficiente y se facilita la comprensión y construcción de código, auqnue finalmente se comprueba que los resultados son idénticos.

3.4. Evaluación del modelo

La fase final del proceso implica la evaluación del modelo k-Nearest Neighbors (k-NN). En primer lugar, se deberá crear el modelo con los hiperparámetros óptimos encontrados durante la búsqueda en grid y ajustarlo con los datos de entrenamiento. Por último, se evaluará su rendimiento con el conjunto de prueba. La accuracy del modelo con los datos de test proporciona una medida objetiva de la capacidad para hacer predicciones precisas. Otra métrica útil para evaluar el modelo final es la sensibilidad (Recall). Esta medida expresa la proporción de verdaderos positivos, es decir, muestra la capacidad del modelo para identificar correctamente a los pacientes que realmente tienen la enfermedad. Es una medida a tener muy en cuenta en este caso práctico, al ser una situación en las que identificar todos los casos positivos es crucial y no se pueden permitir falsos negativos. Su fórmula es la siguiente:

$$\text{Sensibilidad} = \frac{\text{Verdaderos Positivos}}{\text{Verdaderos Positivos} + \text{Falsos Negativos}}$$

```
# Crear un modelo KNN con los mejores hiperparámetros encontrados
best_knn = KNeighborsClassifier(n_neighbors=29, metric='chebyshev', weights='distance')

# Ajustar el modelo con los datos de entrenamiento
best_knn.fit(xtrain_final, y_train)

# Hacer predicciones en el conjunto de prueba
y_pred = best_knn.predict(xtest_final)

# Evaluar el modelo en el conjunto de prueba
test_accuracy = best_knn.score(xtest_final, y_test)
print("Precisión en el conjunto de prueba:", test_accuracy)
```

```
# Calcular la sensibilidad (recall)
sensibilidad = recall_score(y_test, y_pred)
print("Sensibilidad en el conjunto de prueba:", sensibilidad)
```

```
Precisión en el conjunto de prueba: 0.7863247863247863
Sensibilidad en el conjunto de prueba: 0.9655172413793104
```

El modelo devuelve una accuracy del 78.63 %, por tanto, tiene una capacidad de predicción alta, aunque mejorable. Sin embargo, la sensibilidad, una medida específica para problemas sanitarios (en los que se desean minimizar los falsos negativos) devuelve un valor muy alto, del 0.9655. Esto significa que el modelo es capaz de predecir el 96.55 % de verdaderos positivos. Es decir, del total de las personas con enfermedades de hígado, el modelo será capaz de detectarles la enfermedad al 96.55 %.

3.5. Entrenamiento del modelo final

El último paso será la construcción del modelo final y definitivo que se pasase a producción; es decir, el que contenga todos los datos del conjunto. Para ello, será necesario volver a realizar todos los pasos anteriores (preprocesado y ajuste del modelo) sin dividir el conjunto en entrenamiento y test.

```
# Construcción del modelo final

# 1. Preprocesado

# 1.1 Variables categóricas :

def apply_one_hot_encoding(df, factor_columns):
    # Crea el codificador OneHotEncoder
    encoder = OneHotEncoder(sparse=False)

    # Selecciona las columnas de factor o strings
    factor_data = df[factor_columns]

    # Aplica la codificación OneHotEncoder a las columnas de factor
    encoded_data = encoder.fit_transform(factor_data)

    # Crea un nuevo dataframe con las columnas codificadas
    encoded_columns = encoder.get_feature_names_out(factor_columns)
    encoded_df = pd.DataFrame(encoded_data, columns=encoded_columns, index=df.index)

    print(encoded_df.head())

    # Une el nuevo data frame con las columnas codificadas al dataframe original
    final_df = pd.concat([df.drop(factor_columns, axis=1), encoded_df], axis=1)

    return final_df

# Ejectutar función
x_knn = apply_one_hot_encoding(X, factor_columns)


# 1.2 Imputación de valores faltantes
```

```
# Ajustar y transformar los datos con el imputador
x_knn = imputer.fit_transform(x_knn)


# 1.3 Normalizar los datos
x_knn = scaler.fit_transform(x_knn)



# 2. Ajustar el modelo final con todos los datos
best_knn.fit(x_knn, y)
```

Se obtiene el modelo final, ajustado con todos los datos preprocesados, listo para enviar a producción.

4. Construcción de un modelo de árbol de clasificación

A continuación, se ajustará un modelo de árbol de clasificación a los datos, empleando una pipeline para el preprocesamiento de datos y se ajustarán los hiperparámetros utilizando un grid search. De nuevo, se evaluará el rendimiento del modelo utilizando k-fold validation (k=10) y una partición de holdout al 20 %, junto con una métrica adicional para obtener una comprensión más profunda de su capacidad predictiva.

Antes de comenza con el ajuste en Python, se va a presentar una introducción teórica, en la que se entenderá cómo funcionan los árboles y la construcción de los mismos.

4.1. ¿Qué son y cómo funcionan?

Un árbol de decisión, genera un modelo que divide el espacio de los predictores agrupando observaciones con valores similares para la variable respuesta.

Los árboles de decisión están formados por nodos y su lectura se debe de hacer de arriba hacia abajo, estos nodos los podemos dividir en tres tipos:

- **Primer nodo o nodo raíz:** en él se produce la primera división en función de la variable más importante
- **Nodos intermedios:** se encuentran tras la primera división, vuelven a dividir nuestro conjuntos de datos en función de otras variables
- **Nodos terminales u hojas:** se encuentran en la parte inferior de nuestro árbol y su función es indicar la clasificación definitiva

En la figura 3, se muestra gráficamente la fórma de un árbol de clasificación, dónde indentificar las partes anteriores es trivial Para ralizar la clasificación, se aplica el **algoritmo de Hunt** que divide la muestra en subconjuntos buscando una separación óptima.

Destacar que un nodo va a ser terminal si representa a un subconjunto de instancias de la misma clase, en otro caso se seguirá dividiendo en subconjuntos más pequeños, pero **¿en qué nos basamos para dividir en el subconjunto?**

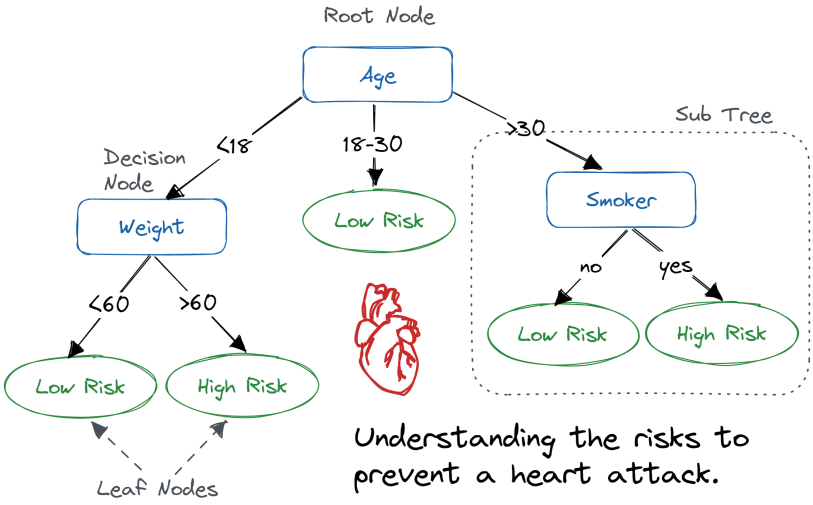

Figura 3: Ejemplo de árbol de clasificación

Para ello tenemos que seleccionar una medida qué nos indique que atributo de separación escoger. Esta medida se maximizará o se minimizará cuando las particiones obtenidas sean muy homogéneas. Se repasan las medidas más utilizadas:

- **Entropía**: mide la impureza de un conjunto de datos o la incertidumbre en la clasificación de las muestras; su fórmula general es la siguiente:

$$H(P) = -\sum_{i=1}^{n} P(x_i) log_2(P(x_i))$$

  dónde

  - $H(X)$: es la entropía del conjunto de datos X
  - $P(x_i)$: es la probabilidad de que una muestra pertenezca a la clase x_i

  El sumatorio se realiza sobre todas las clases x_i presentes en el conjunto de datos.

  El objetivo es encontrar la división que reduce la entropía en los subconjuntos resultantes y, por lo tanto, aumenta la pureza de la clasificación en cada nodo. Este proceso se repite en cada nodo hasta que se alcanza una cierta pureza o se alcanza una profundidad máxima en el árbol.

- **Índice de Gini** : También mide la impureza o desorden. Esta es la medida usada en el algoritmo Rpart, que mide la probabilidad de que una muestra sea clasificada incorrectamente cuando se elige aleatoriamente una etiqueta de acuerdo con la distribución de clases presente en el subconjunto. Su fórmula general es la siguiente:

$$Gini(X) = 1 - \sum_{i=1}^{n} (P(x_i))^2$$

  - $Gini(X)$ es el índice de Gini del conjunto de datos X

- $P(x_i)$ es la probabilidad de que una muestra pertenezca a la clase x_i

Al igual que con la entropía, si se toma esta definición de índice de Gini, lo que se busca es la división que minimice esta medida, ya que en un conjunto de datos perfectamente puro, donde todas las muestras pertenecen a una sola clase, el índice de Gini es 0. Cuanto mayor sea el índice de Gini, mayor será la impureza o la probabilidad de que una muestra se clasifique incorrectamente.

¿Y cuándo se para? Se siguen haciendo subdivisones hasta que se llegue a un subconjunto cuyos datos pertenezcan todos a la misma clase o hasta que el número de datos restantes en ese nodo sea lo suficientemente pequeño, este número se puede indicar a la hora de entrenar el algoritmo, siendo un hiperparámetro.

Repaso del algoritmo de clasificación:

1. Se crean los nodos que minimicen o maximicen la **medida** seleccionada en el algoritmo.
2. Se detiene la construcción del árbol si todos los ejemplos pertenecen a la misma clase o si el número de instancias es inferior al seleccionado.
3. Se siguen creando de manera recursiva tantos sub-árboles como posibles valores tanga el atributo seleccionado.

4.2. Preprocesado de datos

Como bien se ha comentado antes, el preprocesado de los datos es una parte muy importante a la hora de aplicar cualquier algoritmo de aprendizaje automático, ya que dependiendo del que se quiera utilizar, se deben de tomar unas medidas u otras.

En este caso, se quiere aplicar un árbol de clasificación, que es un algoritmos bastante robusto, ya que puede tratar con datos faltantes y no se ve afectado por diferentes escalas. Sin embargo, las variables binarias o categóricas, se tienen que transformar en columnas de 0 y 1, con lo que se suele llamar OneHotEncoding.

No obstante, en esta sección de preprocesado de datos, además de aplicar el OneHotEncoding a las variables categóricas (algo necesario para aplicar el algoritmo), se han estandariazado las variables numéricas y se han imputado los valores faltantes mediante la media, algo que aunque no es necesario, se considera una buena práctica y mejorará el rendimiento del algoritmo.

Para realizar esto en Python, se ha creado una **Pipeline** cómo se puede ver en el siguiente código:

```
# Crear un pipeline que preprocese los datos numéricos
pipeline_cuantitativas = Pipeline([
    ("imputer",SimpleImputer(strategy="mean")), #Imputamos los datos faltantes con la
    media
    ("scaler", StandardScaler())
])

pipeline_categoricas = Pipeline([
    ("encoder", OneHotEncoder()), # Codificar los datos categóricos
])
```

```
Preprocesador = ColumnTransformer([
    ("num", pipeline_cuantitativas, ["Age", "Total_Bilirubin", "Direct_Bilirubin", "
    Alkaline_Phosphotase", "Alamine_Aminotransferase", "Aspartate_Aminotransferase","
    Total_Protiens","Albumin","Albumin_and_Globulin_Ratio"]), # Aplicar el pipeline numé
    rico a las columnas numéricas
    ("cat", pipeline_categoricas, ["Gender"]) # Aplicar el pipeline categórico a las
    columnas_categóricas
])
```

Destacar que se han creado dos pipeline por separado (una para cada tipo de dato) donde se aplican las medidas comentadas anteriormente, para posteriormente unirlas en un nuevo objeto llamado **Preprocesador**. Este es un ColumnTransformer, es decir, que cuando se lo apliquemos a los datos, aplicará cada pipeline a las columnas correspondientes.

Por ahora, sólo se ha creado parte de la pipeline final. Con el siguiente código, se va a añadir el ajuste de un árbol de clasificación, para que cuando se pase la pipeline a los datos, además de preprocesarlos, se ajuste el modelo automáticamente.

```
# Crear un pipeline que preprocese los datos y ajuste un modelo de árbol de decisión
Clasificador_tree = DecisionTreeClassifier(random_state=10453131) # Crear un modelo de á
    rbol de decisión

tree_pipeline = Pipeline([
    ("preprocessor", Preprocesador), # Preprocesar los datos con el transformador de
    columnas
    ("Clasificador_tree", Clasificador_tree) # Ajustar un modelo de árbol de decisión
])
```

Con esto se consigue que con una simple línea de código, se preprocesen los datos y se ajuste un modelo de árbol de clasificación. El siguiente paso es ajustar los hiperparámetros de este modelo que se está ajustando, algo que se va a realizar en la siguiente sección.

4.3. Ajuste de hiperparámetros

En primer lugar, se van a explicar de forma breve los hiperparámetros que se ajustarán:

- **max.depth**: Profundidad máxima de nuestro árbol, acotamos mediante este hiperparámetro la cantidad máxima de nodos que puede haber en una rama
- **min.samples.split**: Número mínimo de muestras requeridas para dividir un nodo interno
- **min.samples.leaf**: Número mínimo de muestras requeridas para ser una hoja
- **criterion**: Criterio usado para dividir

Además de estos hiperparámetros, se puede ajustar alguno más como puede ser **max.features**. En la página web de scikit-learn se pueden encontrar todos los hiperparámetros válidos para esta función.

Es fácil ver que al estar estimando cuatro hiperparámetros, el número de combinaciones entre todos

ellos es bastante alto. Se podrían ajustar con bucles como se ha procedido en el k-NN, no obstante, se va a hacer uso de un **GridSearch**, el cual se va a encargar de ajustar todas las combinaciones posibles de hiperparámetros y devolverá la mejor según la métrica y la metodología que se le haya indicado.

En el caso del árbol de clasificación, se ha procedido dividiendo la muestra total en en dos subconjuntos, uno de test y otro de entrenamiento. Este último es el utilizado en esta sección de ajuste de hiperparámetros.

Partiendo de la muestra de entrenamiento, para el ajuste de hiperparámetros se ha aplicado validación cruzada, concretamente un kfold con K=10, es decir el modelo se entrena y evalúa 10 veces, utilizando cada una de las 10 partes como conjunto de prueba una vez y las otras 9 como conjunto de entrenamiento en cada iteración. Esto ayuda a evaluar la capacidad de generalización de un modelo al proporcionar métricas de rendimiento (en este caso se ha usado la accuracy) promediadas sobre las 10 iteraciones, lo que reduce el riesgo de sobreajuste y mejora la confiabilidad de las evaluaciones del modelo.

Una vez entendido cómo se va a proceder para el ajuste de los hiperparámetros, se aplicará en Python con el siguiente código:

```
param_grid_tree = {
    "Clasificador_tree__max_depth":list(range(4, 11, 1)), # Profundidad máxima del árbol
    la miramos de 4 a 10
    "Clasificador_tree__min_samples_split": [10, 15, 20], # Número mínimo de muestras
    requeridas para dividir un nodo interno
    "Clasificador_tree__min_samples_leaf": [10, 15, 20], # Número mínimo de muestras
    requeridas para ser una hoja
    "Clasificador_tree__criterion": ["gini", "entropy"], # Criterio usado para dividir
}
grid_search = GridSearchCV(tree_pipeline, param_grid_tree, cv=10, scoring='accuracy',
    verbose=1) #Creamos y ajustamos el grid con k-fold
```

Destacar que se ha creado un grid con todas las posibles combinaciones de los hiperparámetros. A estos últimos se les ha asignado un rango de valores posibles (acotando sus dominios). Para la profundidad máxima se va a comprobar de 4 a 11. Un árbol con profundidad menor a 4 es demasiado sencillo, mientras que superior a 11 es demasiado complejo considerando el conjunto de datos con el que se está trabajando. En cuanto al criterio se han definido los dos explicados en el punto 4.1, es decir, Entropía y Gini. Por último, en relación al número de observaciones mínimas para dividir un nodo interno y para ser una hoja, se ha probado con 5 valores que van desde 5 hasta 25, un rango que se considera bastante amplio para el número de observaciones con las que se está trabajando.

Una vez se ha entendido el por qué de estos hiperparámetros, con el siguiente código, se divide la muestra original en train y test con un holdout al 20 %, para luego aplicar el grid search a los datos en entrenamiento. Además se pide que devuelva los mejores hiperparámetros y la accuracy media del modelo ajustado.

```
# Dividir aleatoriamente el conjunto de datos en entrenamiento y prueba con Holdout 20%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
    =100453131)

#Ajustamos los hiperparámetros con los datos de entrenamiento
grid_search.fit(X_train,y_train)
```

```
# Mostrar los mejores hiperparámetros encontrados
print("Los mejores hiperparámetros son:", grid_search.best_params_)
print("La precisión del modelo es", round(grid_search.best_score_,4))
```

```
Fitting 10 folds for each of 350 candidates, totalling 3500 fits
Los mejores hiperparámetros son: {'Clasificador_tree__criterion': 'gini', '
    Clasificador_tree__max_depth': 4, 'Clasificador_tree__min_samples_leaf': 5, '
    Clasificador_tree__min_samples_split': 20}
La precisión del modelo es 0.7233
```

El ajuste de hiperparámetros ha devuelto siguiente modelo

- **max.depth**: 4
- **min.samples.split**: 20
- **min.samples.leaf**: 5
- **criterion**: Gini
- **Precisión**: 0.7233

Una vez se tiene el modelo con el mejor ajuste, el siguiente paso es evaluarlo con la muestra de test, que se realizará en el siguiente punto.

4.4. Evaluación del modelo

En esta sección, se va a evaluar el modelo estimado en el apartado anterior. Para ello se utilizan los datos de test, divididos anteriormente. La finalidad de esta evaluación es observar si el modelo generaliza bien y es capar de predecir de forma correcta datos con los que no ha sido entrenado, algo muy importante, ya que es la finalidad de estos modelos de clasificación.

```
accuracy = grid_search.best_estimator_.score(X_test, y_test) #Calculamos la accuracy
print(f"Precisión del modelo en datos de test: {accuracy:.4f}") #La mostramos

# Hacer predicciones en el conjunto de test
y_pred = grid_search.best_estimator_.predict(X_test)

# Calcular la sensibilidad (recall)
sensibilidad = recall_score(y_test, y_pred)
print("Sensibilidad en el conjunto de test:", round(sensibilidad,4))
```

```
Precisión del modelo en datos de test: 0.6752
Sensibilidad en el conjunto de prueba: 0.939
```

El modelo devuelve una accuracy del 67.52 %, por tanto, el modelo generaliza bien (pérdida del 5 % aprox) pero en general su precisión es bastante baja, fallando aproximadamente 3 de cada 10 predicciones. Sin embargo, la sensibilidad, una medida específica para problemas sanitarios (en los que se desean minimizar los falsos negativos) devuelve un valor bastante alto, del 0.939. Esto significa que el modelo es capaz de predecir el 93.9 % de verdaderos positivos. Es decir, del total de las personas con enfermedades de hígado, el modelo será capaz de detectarles la enfermedad al 93.9 %.

4.5. Entrenamiento del modelo final

Una vez se tiene el modelo final con su error asociado, antes de pasarlo a producción para predecir nuevas observaciones, se debe volver a entrenarlo con la muestra completa, ya que así se le aportan toda la información de los datos iniciales. Destacar que el error de este modelo es el conseguido al evaluarlo con los datos de test, por lo que con este nuevo modelo, no se realizará ningun tipo de evaluación, solamente se entrenará y se pasará a "producción", ya listo para intentar predecir las nuevas observaciones que lleguen.

Para entrenar el modelo final, se crea una nueva pipeline, donde se introducirá el algortimo con los hiperparámetros marcados de base. El preprocesado, de la misma manera que en las secciones anteriores, se realiza con el siguiente código:

```
Arbol_Hiperparametros = DecisionTreeClassifier(max_depth=4,min_samples_leaf=15,
    min_samples_split=10,criterion='gini')

tree_pipeline_final = Pipeline([
    ("preprocessor", Preprocesador), # Preprocesar los datos con el transformador de
    columnas
    ("Clasificador_tree", Arbol_Hiperparametros) # Ajustar un modelo de árbol_de_decisión
]) # Creamos otra pipeline que tenga el nuevo modelo, pero que nos limpie los datos de la
    forma adecuada

modelo_final_tree = tree_pipeline_final.fit(X,y) #Ajustamos el modelo
```

Con esto se termina la sección del árbol de clasificación y se tendría el modelo listo para producción.

5. Construcción de un modelo RandomForest

5.1. ¿Qué es y cómo funciona?

La idea detrás del RandomForest, está basada en el Ensemble Learning que es un enfoque en el aprendizaje automático donde se combinan múltiples modelos o algoritmos para mejorar el rendimiento predictivo y la precisión en comparación con un solo modelo. En lugar de depender de un solo modelo, se utilizan varios modelos en conjunto para aprovechar sus puntos fuertes y mitigar sus debilidades.

Funciona al combinar múltiples árboles de decisión para tomar decisiones más precisas y robustas. Cada árbol se entrena con una muestra aleatoria de datos y características, lo que reduce el riesgo de sobreajuste.

El algoritmo toma una decisión de clasificación al permitir que cada árbol emita su propia predicción y luego elige la clase más votada entre los árboles. Esto hace que el Random Forest sea eficaz, resistente al ruido y capaz de manejar características importantes y no importantes.

5.2. Preprocesado y ajuste de hiperparámetros

El preprocesado detrás de este modelo, es el mismo que el usado en los árboles de clasificación, por lo que se parte de la misma pipeline y con el mismo procedimiento empleado en el punto **4.2**. Lo único que cambia es que ahora en vez de usar "DecisionTreeClassifier"que ajustaba un árbol de clasificación, se usa RandomForestClassifier"que ajusta un bosque de clasificación.

```
# Crear un pipeline que preprocese los datos y ajuste un modelo de bosque de clasificación
Clasificador_forest = RandomForestClassifier(random_state=100453131) # Crear un modelo de
    bosque aleatorio

forest_pipeline = Pipeline([
    ("preprocessor", Preprocesador), # Preprocesar los datos con el transformador de
    columnas
    ("Clasificador_forest", Clasificador_forest) # Ajustar un modelo de bosque aleatorio
])
```

El siguiente paso es ajustar los hiperparámetros del modelo, para ello, al igual que en con los árboles de clasificación, se realiza primero un holdout al 20 % para la validación outer y un Kfold con K=10 para la validación inner. Si se quiere desarrollar más sobre este punto, se recomienda mirar el punto **4.3**.

Dentro del RandomForest, se pueden ajustar muchos hiperparámetros, la gran mayoría de ellos iguales que los del árbol de clasificación, pero quizá el más importate es el asociado al número de árboles del bosque. Se puede inferir que el coste computacional detrás de ajustar un RandomForest, es mucho más alto que el de otros algoritmos, por lo que solo se va a ajustar el número de árboles del bosque. Se realiza al igual que en el punto **4.3** con un GridSearch con el siguiente código.

```
param_grid_forest = {
    "Clasificador_forest__n_estimators": [5,10,20,30,40,50,60,70,80,90,100],  # Número de
    árboles estimados
}
grid_search = GridSearchCV(forest_pipeline, param_grid_forest , cv=10, scoring='accuracy',
    verbose=1) #Creamos y ajustamos el grid con k-fold

# Dividir aleatoriamente el conjunto de datos en entrenamiento y prueba con Holdout 20%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
    =100453131)

#Ajustamos los hiperparámetros con los datos de entrenamiento
grid_search.fit(X_train,y_train)

# Mostrar los mejores hiperparámetros encontrados
print("Los mejores hiperparámetros son:", grid_search.best_params_)
print("La precisión del modelo es", round(grid_search.best_score_,4))
```

```
Fitting 10 folds for each of 11 candidates, totalling 110 fits
Los mejores hiperparámetros son: {'Clasificador_forest__n_estimators': 50}
La precisión del modelo es 0.7319
```

Por lo que el ajuste de hiperparámetros ha devuelto

- **n.estimators**: 50
- **Precisión**: 0.7319

5.3. Evaluación del modelo

Al igual que con los modelos anteriores, se usa el modelo con los mejores hiperparámetros estimados, y se le aplica la validación outer con los datos de test, para ver si generaliza bien.

```
accuracy = grid_search.best_estimator_.score(X_test, y_test) #Calculamos la accuracy
print(f"Precisión del modelo en datos de test: {accuracy:.4f}") #La mostramos

# Hacer predicciones en el conjunto de prueba
y_pred = grid_search.best_estimator_.predict(X_test)

# Calcular la sensibilidad (recall)
sensibilidad = recall_score(y_test, y_pred)
print("Sensibilidad en los datos de test:", round(sensibilidad,4))
```

```
Precisión del modelo en datos de test: 0.6752
Sensibilidad en el conjunto de test: 0.9024
```

El modelo devuelve una accuracy del 67.52 %, por tanto, el modelo generaliza bien (pérdida del 6 % aprox) pero en general su precisión es bastante baja, fallando aproximadamente 3 de cada 10 predicciones. Sin embargo, la sensibilidad, una medida específica para problemas sanitarios (en los que se desean minimizar los falsos negativos) devuelve un valor bastante alto, del 0.9024. Esto significa que el modelo es capaz de predecir el 90.24 % de verdaderos positivos. Es decir, del total de las personas con enfermedades de hígado, el modelo será capaz de detectarles la enfermedad al 90.24 %.

5.4. Entrenamiento del modelo final

Al igual que se hizo con los modelos anteriores, se vuelve a entrenar con todos los datos, antes de pasarlo a producción, por lo que se actúa forma equivalente a las secciones anteriores:

```
    Bosque_Hiperparametros = RandomForestClassifier(n_estimators=50)

Bosque_pipeline_final = Pipeline([
    ("preprocessor", Preprocesador), # Preprocesar los datos con el transformador de
    columnas
    ("Clasificador_forest", Clasificador_forest) # Ajustar un modelo de bosque aleatorio
]) #Creamos otra pipeline que tenga el nuevo modelo, pero que nos limpie los datos de la
    forma adecuada

modelo_final_forest = Bosque_pipeline_final.fit(X,y) #Ajustamos el modelo
```

Una vez que se tienen los tres modelos estimados, resulta interesante realizar una comparación, para ver cuál es mejor de los tres y tomar una decisión final. Esto se va a realizar en la siguiente sección.

6. Comparación de los tres métodos

En este punto se llevará a cabo una comparación exhaustiva entre tres modelos de clasificación ampliamente utilizados: el K-Nearest Neighbors (KNN), el Árbol de Clasificación y el Random Forest. Cada uno

de estos modelos ha sido previamente estimados y comentados en secciones anteriores.

El objetivo de esta comparación es evaluar y analizar el rendimiento de estos modelos en términos de precisión, robustez y capacidad de generalización en un contexto de clasificación de datos.

A lo largo de esta sección, se explorarán los resultados anteriores y recalcaremos las fortalezas y debilidades de cada modelo en función de métricas de evaluación específicas (precisión y sensibilidad). Esta comparación ayudará a determinar cuál de estos modelos es el más adecuado para abordar el problema de clasificación. Se muestra un resumen de los resultados de las secciones anteriores en la siguiente tabla.

Modelo	Precisión	Sensibilidad
K-Nearest Neighbors (KNN)	78.63 %	96.55 %
Árbol de Clasificación	67.52 %	93.9 %
Random Forest	67.52 %	90.24 %

Se puede observar cómo se tiene un claro ganador. En términos de precisión y sensibilidad, el k-NN es el modelo que mejor capacidad predictiva tiene, superando a los otros dos en aproximadamente un 11 %, por lo que para este problema de clasificación, entre los tres algoritmos empleados, se elegirá el k-NN para hacer futuras clasificaciones, por su superioridad a la hora de clasificar las observaciones en general, pero además por ser el que mejor sensibilidad tiene, es decir, de identificar a los positivos reales, función para lo que se utilizará.

Por otra parte, el Árbol de clasificación y Random Forest, devuelven la misma precisión, incluso sorprende que el Random Forest tenga una sensibilidad menor, ya que este es el conjunto de muchos árboles de clasificación, lo que puede llegar a hacer ver, que los árboles de clasificación no se ajustan bien a nuestros datos, y aunque los combinemos no son capaces de sacar más información para clasificar mejor. Mencionar además que ninguno de los tres modelos nos deja una interpretación clara y sencilla, por lo que este criterio no ha sido utilizado a la hora de tomar decisiones.

Como continuación al trabajo y de cara a mejorar los resultados predictivos, sería interesante, explorar otros algoritmos de clasificación, como máquinas vector soporte (SVM), redes neuronales, o métodos de clasificación más avanzados, como pueden ser los basados en el ensemble learning, pero combinando otros métodos que no sean los árboles, ya que se ha visto que no se ajustan muy bien a los datos.

7. Valoración personal

Como nuestra primera experiencia con los algorimos k-NN, Árbol de clasificación y Random Forest así como con Python en el aprendizaje automático, hemos abordado un trabajo desafiante. En nuestro proceso de aprendizaje, hemos asumido el reto de aprender tres nuevos algoritmos así como mejorar nuestros conocimientos en nuevo lenguaje de programación como es Python, al que no estábamos acostumbrados. Hemos desarrollado un código funcional para aplicar estos algoritmos a un problema específico y hemos trabajado en la optimización de hiperparámetros. Finalmente hemos comprendido los conceptos fundamentales de estos tres algortimos y cómo implementarlos en Python.
Hemos explorado la búsqueda de hiperparámetros a través de la búsqueda en grid y también mediante un enfoque más simple de ajuste de hiperparámetros. Ambos métodos son valiosos y nos han permitido

encontrar la configuración óptima del modelo.
En general, nuestro enfoque inicial en el aprendizaje automático con Python nos entusiasma. Seguiremos explorando, practicando y abordando nuevos proyectos para seguir mejorando nuestras habilidades. El aprendizaje automático puede ser desafiante, pero con paciencia y perseverancia, podemos lograr resultados significativos.