

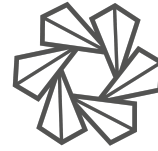
1 Portada



UNIVERSIDAD
AUTÓNOMA
DE QUERÉTARO



FACULTAD
DE INGENIERÍA



DIPFI
POSGRADO
INGENIERÍA

Universidad Autónoma de Querétaro

Maestría en Ciencias en Inteligencia Artificial

Materia - Tópicos Selectos IV - Machine Learning

Actividad - Árbol de decisión

Profesor - Dr. Marco Aceves

Alumno - Juan Ignacio Ortega Gómez

Expediente - 309236

Quertearo, Queretaro a 10 de Junio de 2022

2 Introducción

La inferencia inductiva es el proceso de pasar de ejemplos concretos a modelos generales. En una forma, el objetivo es aprender a clasificar objetos o situaciones mediante el análisis de un conjunto de instancias cuyas clases se conocen [6].

Las técnicas de minería de datos utilizan básicamente el algoritmo ID3 ya que es el algoritmo básico de clasificación. Milija Suknovic (et.al) identificó algunos componentes reutilizables a partir del algoritmo ID3. Proponen un marco de árbol de decisiones genérico que admite el diseño de componentes reutilizables. El marco de árbol de decisión genérico propuesto consta de varios subproblemas que se reconocieron mediante el análisis de algoritmos de inducción de árbol de decisión bien conocidos, a saber, ID3, C4.5, CART, CHAID, QUEST, GUIDE, CRUISE y CTREE [2].

Un árbol de decisión es un formalismo para expresar tales mapeos. Un árbol es un nodo de hoja etiquetado con una clase o una estructura que consta de un nodo de prueba vinculado a dos o más subárboles. Un nodo de prueba calcula algún resultado en función de los valores de atributo de una instancia, donde cada posible resultado está asociado con uno de los subárboles [6].

En este trabajo se propone un algoritmo basado en ID3 para genera un árbol de decisión, el cual es probado utilizando una base de datos para clasificar los tipos de medicamentos que necesita un paciente, además, los resultados son probados mediante los rendimientos de su clasificación.

3 Marco teórico

3.1 Árboles de decisión

3.1.1 Definición

Los árboles de decisión (DT) son un método de aprendizaje supervisado no paramétrico que se utiliza para la clasificación y la regresión. El objetivo es crear un modelo que prediga el valor de una variable de destino mediante el aprendizaje de reglas de decisión simples deducidas de las características de los datos. Un árbol puede verse como una aproximación constante por partes [1].

3.1.2 Ventajas y Desventajas

Algunas ventajas de los árboles de decisión son:

- Fácil de entender y de interpretar. Los árboles se pueden visualizar.
- Requiere poca preparación de datos. Otras técnicas a menudo requieren la normalización de datos, es necesario crear variables ficticias y eliminar valores en blanco. Sin embargo, tenga en cuenta que este módulo no admite valores faltantes.
- El costo de usar el árbol (es decir, predecir datos) es logarítmico en la cantidad de puntos de datos usados para entrenar el árbol.
- Capaz de manejar datos numéricos y categóricos. Sin embargo, la implementación de scikit-learn no admite variables categóricas por ahora. Otras técnicas suelen estar especializadas en analizar conjuntos de datos que tienen un solo tipo de variable. Ver algoritmos para más información.
- Capaz de manejar problemas de múltiples salidas.

- Utiliza un modelo de caja blanca. Si una situación dada es observable en un modelo, la explicación de la condición se explica fácilmente mediante lógica booleana. Por el contrario, en un modelo de caja negra (por ejemplo, en una red neuronal artificial), los resultados pueden ser más difíciles de interpretar.
- Posibilidad de validar un modelo mediante pruebas estadísticas. Eso permite dar cuenta de la fiabilidad del modelo.
- Tiene un buen desempeño incluso si sus supuestos son algo violados por el verdadero modelo a partir del cual se generaron los datos.

Las desventajas de los árboles de decisión incluyen:

- Los aprendices de árboles de decisión pueden crear árboles demasiado complejos que no generalizan bien los datos. Esto se llama sobreajuste. Para evitar este problema, son necesarios mecanismos como la poda, establecer el número mínimo de muestras requeridas en un nudo de la hoja o establecer la profundidad máxima del árbol.
- Los árboles de decisión pueden ser inestables porque pequeñas variaciones en los datos pueden generar un árbol completamente diferente. Este problema se mitiga mediante el uso de árboles de decisión dentro de un conjunto.
- Las predicciones de los árboles de decisión no son uniformes ni continuas, sino aproximaciones constantes por partes, como se ve en la figura anterior. Por lo tanto, no son buenos para la extrapolación.
- Se sabe que el problema de aprender un árbol de decisión óptimo es NP-completo bajo varios aspectos de optimización e incluso para conceptos simples. En consecuencia, los algoritmos prácticos de aprendizaje del árbol de decisiones se basan en algoritmos heurísticos, como el algoritmo voraz, en el que se toman decisiones localmente óptimas en cada nodo. Dichos algoritmos no pueden garantizar la devolución del árbol de decisión globalmente óptimo. Esto se puede mitigar entrenando varios árboles en un alumno de conjunto, donde las características y las muestras se muestrean aleatoriamente con reemplazo.
- Hay conceptos que son difíciles de aprender porque los árboles de decisión no los expresan fácilmente, como XOR, problemas de paridad o multiplexor.
- Los aprendices de árboles de decisión crean árboles sesgados si dominan algunas clases. Por lo tanto, se recomienda equilibrar el conjunto de datos antes de ajustarlo al árbol de decisión [1].

3.1.3 Cálculos paso a paso:

Paso 1 - Calcular la entropía de los atributos y escoger el mayor como raíz.

Paso 2 - Calcular la ganancia de los atributos restantes y encontrar cuál atributo es el siguiente nodo.

Paso 3 - Encontrar un nodo por cada rama de la raíz si es que aún no se ha llegado a clasificaciones separadas en todas las ramas.

Paso 3 - Seguir el proceso de selección de nodos hasta que se terminen los atributos o se tenga clasificaciones separadas en cada rama [2].

3.2 K-Fold

K-Fold, también llamado validación cruzada, es un procedimiento que consiste en partir en k veces los datos y realizar pruebas con cada k -partición.

El parámetro K indica cuántas veces se tiene que particionar los datos. Los K más utilizados son 3, 5 y 10 particiones. La K se suele reemplazar por el número de k -particiones, ‘10-Fold’.

El método es muy popular porque los resultados tienen un menor sesgo y se realiza una estimación menos optimista del rendimiento y la exactitud de un algoritmo.

El algoritmo general es el siguiente:

1. Se desordenan los datos de manera aleatoria.
2. Se separan los datos en k grupos. En este caso, se recomienda crear copias de los datos, cada una con la partición que le corresponde. Por ejemplo, si se quiere hacer un ‘5-Fold’, se hacen cinco copias con los datos. En la primera copia se quita la primera quinta parte, en la segunda copia, la segunda quinta parte y así sucesivamente hasta completar las cinco copias.
3. Las partes que se quitaron serán los bancos de pruebas. La primera prueba se realiza con el modelo de la primera copia y los datos de prueba que se quitaron del mismo y así sucesivamente.
4. Se guardan las métricas de calidad (error, tasa de clasificación, exactitud, precisión, sensibilidad y puntaje F-beta).
5. Se desecha el modelo y el banco de pruebas con el que se realizó.
6. Se cambia al siguiente modelo con el siguiente banco de pruebas, se repite el procedimiento desde el paso cuatro hasta que se terminen las pruebas con todas las particiones.

Los datos se dividen lo más homogéneamente posible, es decir, que cada partición contenga aproximadamente la misma cantidad de datos, como forma gráfica de representar esto se puede mostrar el ejemplo de la siguiente figura [5]:

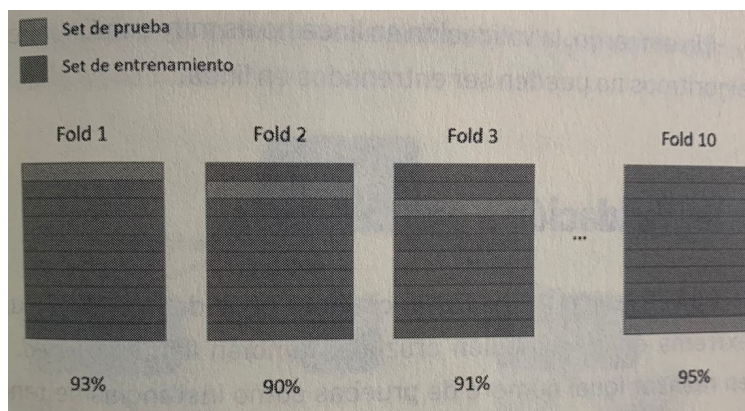


Figura 1. Ejemplo de la distribución de las pruebas para una validación cruzada ‘10-Fold’ [5].

Figura 1. Ejemplo de la distribución de las pruebas para una validación cruzada ‘10-Fold’ [5].

3.3 Métricas de rendimiento

3.3.1 MSE

Es probablemente el método más comúnmente utilizado para poder predecir el error. Permite evaluar el rendimiento de múltiples modelos en un problema de predicción cuando se trata de datos continuos.

Varía en un rango de $[0, inf]$, siendo menor valor de MSE, un mejor rendimiento del modelo.

Su formulación es la siguiente:

$$MSE = \frac{(p_1 - a_1)^2 + \dots + (p_n - a_n)^2}{n}$$

Donde ‘a’ es el valor real actual y ‘p’ es el valor predicho [5].

3.3.2 Tasa de clasificación

La manera más simple de evaluar un modelo con características nominales y discretas es por medio de la tasa de clasificación, cuya formulación es la siguiente [5]:

$$Tasadeclasificacin = 1 - \frac{Clasificacionesincorrectas}{Totaldeprediccionesrealizadas}$$

3.3.3 Matriz de confusión binaria

Solo puede haber cuatro diferentes tipos de resultados que arrojen la matriz de confusión, mostrada en la fig. 2:

- Verdadero positivo (TP) - Se espera que tenga in valor positivo de su carcterística y se obtiene un valor positivo también.
- Verdadero negativo (TN) - Se espera un valor negativo de su carcaterística y se predice un valor negativo también.
- Falso positivo (FP) - Se tiene un valor negativo a pesar de haber predicho un valor positivo.
- Falso negativo (FN) - Se tiene un valor positivo a pesar de haber predicho un valor negativo [5].

		Predicción	
		Positivos	Negativos
Observación	Positivos	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Figura 2. Matriz de confusión para cálculo del error [5].

Figura 2. Matriz de confusión para cálculo del error [5].

3.3.4 Exactitud

Se puede calcular la tasa de clasificación de una manera más precisa con la matriz de confusión. Se puede calcular como una métrica de exactitud del modelo, cuya formulación es la siguiente [5]:

$$Exactitud = \frac{(TP+TN)}{(TP+TN+FP+FN)}$$

3.3.5 Precisión

Se puede definir como la tasa de las muestras predichas que son relevantes, se calcula como sigue [5]:

$$Precisión = \frac{TP}{TP+FP}$$

3.3.6 Sensitividad (Recall)

La sensibilidad, también llamado recall, se puede definir como la tasa de las muestras seleccionadas que son relevantes a la prueba. Se obtiene de la siguiente manera:

$$Sensitividad = \frac{TP}{TP+FN}$$

3.3.7 Puntaje F1

El puntaje F1 (F1 Score en inglés), puntaje F-beta con una beta de 1, se puede definir como la media armónica entre recall y precisión, y se calcula como se muestra en la siguiente ecuación:

$$PuntajeF1 = \frac{2*TP}{2*TP+FP+FN}$$

3.4 Conceptos en el manejo de datos

3.4.1 Cuartiles

La mediana divide la muestra a la mitad, los cuartiles lo dividen, tanto como sea posible, en cuartos.

$$Primercuartil = 0.25 * (n + 1)$$

$$Segundocuartil = 0.5 * (n + 1) \rightarrow \text{Idéntico a la mediana}$$

$$Tercercuartil = 0.75 * (n + 1)$$

El resultado te dice el número del valor que representa el X cuartil, de los datos ordenados de forma ascendente.

Solo si el resultado es un entero, si no, se toma el promedio de los valores de la muestra de cualquier lado de este valor, tomando la muestra de forma ordenada ascendente [5].

3.4.2 Normalización de datos

Algunos algoritmos de inteligencia artificial requieren que todos los datos se centren en un rango específico de valores, normalmente de -1 a 1 o de 0 a 1 . Incluso si no se requiere que los datos se encuentren dentro de los valores, es buena idea generalmente asegurarse de que los valores se encuentran dentro de un rango específico.

Normalización de valores ordinales Para normalizar un set ordinal, se tiene que preservar el orden.

Normalización de valores cuantitativos Lo primero que se tiene que hacer es observar el rango en el cual se encuentran dichos valores y el intervalo al que se quiere normalizar. No todos los valores requieren ser normalizados.

Es necesario realizar los cálculos de las siguientes variables para encontrar el valor normalizado:

1. $Mx_{modelosdatos} = el\ valorms\ altodelaobservacinsinnormalizar$
2. $Mn_{modelosdatos} = elmenoraltodelaobservacinsinnormalizar$
3. $Mx_{monormalizado} = el\ valorms\ altolimtrofealqueelmx_{modelosdatos} sernormalizado$
4. $Mn_{monormalizado} = El\ valorms\ bajolimtrofealqueelmn_{modelosdatos} sernormalizado$
5. $Rangodespacedatos = Mx_{modelosdatos} - Mn_{modelosdatos}$
6. $Rangonormalizado = Mx_{monormalizado} - Mn_{monormalizado}$
7. $D = Valoranormalizar - Mn_{modelosdatos}$
8. $DPct = \frac{D}{Rangodedatos}$
9. $dNorm = Rangonormalizado * DPct$
10. $Normalizado = Mn_{monormalizado} + dNorm$

De esta forma se obtiene el valor normalizado [5].

3.5 Intervalo de confianza

3.5.1 Definición

Un intervalo de confianza es un rango de valores en el que estamos bastante seguros de que reside nuestro verdadero valor [4].

3.5.2 Cálculo del intervalo de confianza

Paso 1: Empieza con - El número de observaciones n - La media X - La Desviación Estándar s

Paso 2: decide qué intervalo de confianza quieres: 95% o 99% son opciones comunes. Luego encuentra el valor “Z” para ese intervalo de confianza aquí:

Intervalo de Confianza de:

80% - Z = 1,282

85% - Z = 1,440

90% - Z = 1,645

95% - Z = 1,960

99% - Z = 2,576

99,5% - Z = 2,807

99,9% - Z = 3,291

Paso 3: usa ese valor Z en esta fórmula para el intervalo de confianza

$$X \pm (Z * s / \text{raíz}(n))$$

Donde: - X es la media - Z es el valor Z elegido de la tabla anterior - s es la desviación estándar - n es el número de observaciones

El valor después del \pm se llama margen de error [4].

3.6 Base de datos

Se utilizó la base de datos (DB por sus siglas en inglés) Drugs A, B, C, X, Y obtenida de Kaggle, donde se han recopilado datos sobre un conjunto de pacientes, todos los cuales padecían la misma enfermedad. Durante su curso de tratamiento, cada paciente respondió a uno de los 5 medicamentos, Medicamento A, Medicamento B, Medicamento c, Medicamento x e y.

Parte de su trabajo es construir un modelo para averiguar qué medicamento podría ser apropiado para un futuro paciente con la misma enfermedad. Las características de este conjunto de datos son la edad, el sexo, la presión arterial y el colesterol de los pacientes, y el objetivo es el fármaco al que respondió cada paciente [3].

4 Metodología

4.1 Recolección de datos

Importamos librerías necesarias durante el desarrollo del algoritmo.

```
[ ]: import pandas as pd
import seaborn as sns
import numpy as np
import math as mt
import random as rd
from collections import Counter
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
```

Accedemos a la carpeta de drive donde se encuentra la base de datos.

```
[ ]: from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

Cargamos la base de datos Drugs ABCXY, dentro de la variable 'DB'.

```
[ ]: DB = pd.read_csv('/content/drive/MyDrive/ML_TSIV/Decision_Tree/Drugs ABCXY.csv')
```

Dejamos estática la base de datos sin modificar procesos posteriores.

```
[ ]: DB_STAY = DB
```

Damos un primer vistazo a la base de datos, mostrando los datos organizados en una tabla.


```
[ ]: DB.head()
```

```
[ ]:   Age Sex    BP Cholesterol Na_to_K Drug
0    23  F   HIGH          HIGH  25.355 drugY
1    47  M   LOW          HIGH  13.093 drugC
2    47  M   LOW          HIGH  10.114 drugC
3    28  F  NORMAL          HIGH   7.798 drugX
4    61  F   LOW          HIGH  18.043 drugY
```

Función para transformar cada atributo cualitativo dentro de la base de datos a datos numéricos.

```
[ ]: def Cualit2Num(DBCN):
    Titles = list(DBCN.columns)

    Titles_str = []
    for title in Titles:
        if type(DBCN[title][0]) == str:
            Titles_str.append(title)

    for title in Titles_str:
        types = pd.unique(DBCN[title])
        i = 0
        for data in types:
            data_loc = np.where(DBCN[title] == data)[0]
            for pos in data_loc:
                DBCN[title][pos] = i
            i += 1
    return(DBCN)
```

Llamamos a la función ‘Cualit2Num’ anterior para transformar la base de datos.

```
[ ]: DB = Cualit2Num(DB)
DB.head()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:15:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

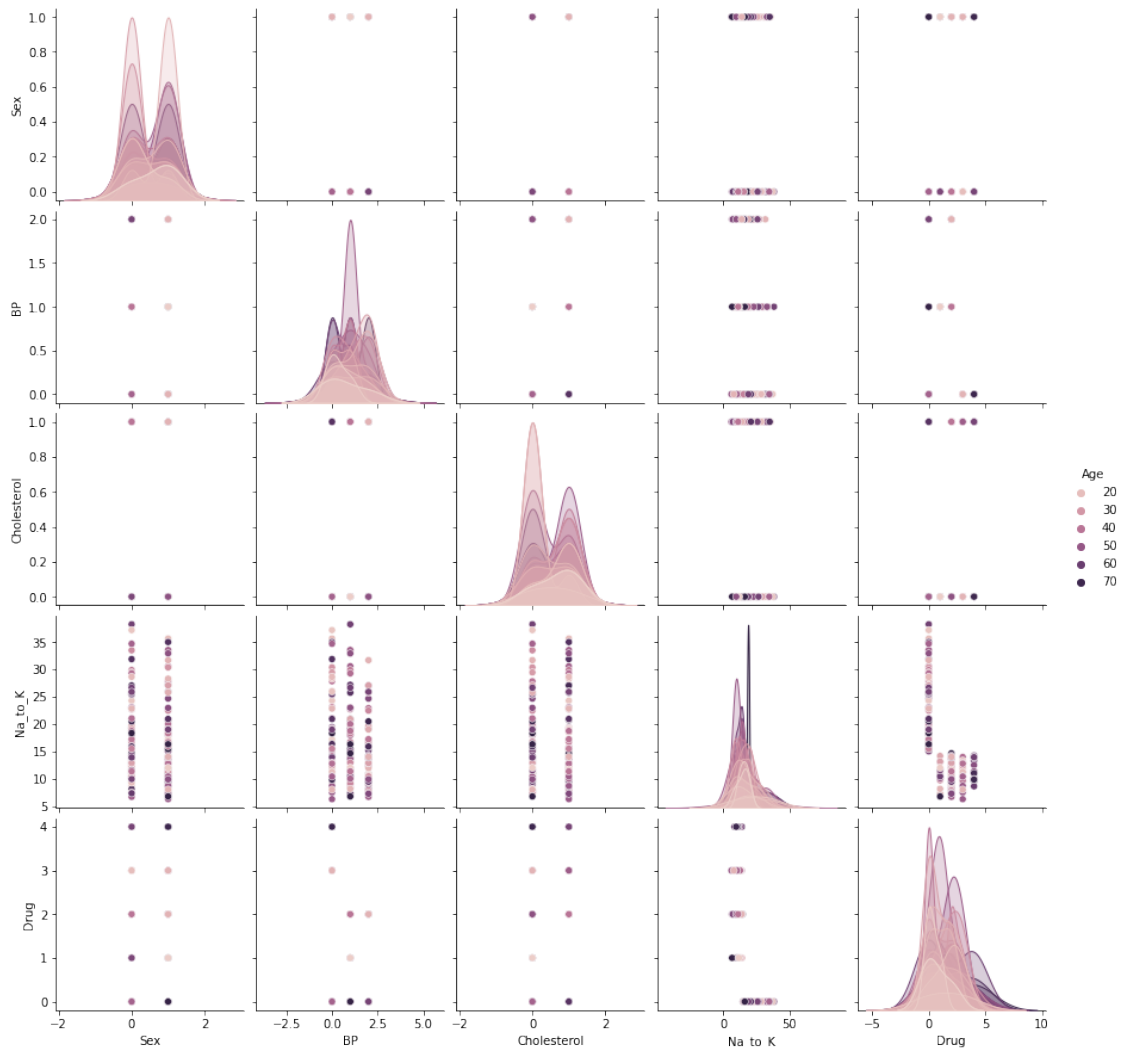
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
from ipykernel import kernelapp as app

```
[ ]:   Age Sex BP Cholesterol Na_to_K Drug
0    23  0  0          0  25.355  0
1    47  1  1          0  13.093  1
2    47  1  1          0  10.114  1
3    28  0  2          0   7.798  2
4    61  0  1          0  18.043  0
```

Posteriormente, desplegamos una representación gráfica de los datos, generando las relaciones entre cada atributo.

```
[ ]: sns.pairplot(DB, vars = DB.columns[1:6], hue = DB.columns[0])
```

```
[ ]: <seaborn.axisgrid.PairGrid at 0x7ff8e66e0590>
```



4.2 Preparación de los datos

4.2.1 Analizamos los datos de forma elemental.

Calculamos la cantidad de Atributos e Instancias.

```
[ ]: NoAtributos = len(DB.T)
     NoInstancias = len(DB)
```

Convertimos la base de datos en un arreglo.

```
[ ]: DBar = DB.to_numpy()
```

Calculamos el máximo y mínimo de cada Atributo.

```
[ ]: MaximoDeAtributos = []
MinimoDeAtributos = []
for idx in range(NoAtributos):
    CaractMax = max(DBar.T[idx])
    CaractMin = min(DBar.T[idx])
    MaximoDeAtributos.append(CaractMax)
    MinimoDeAtributos.append(CaractMin)
```

Calculamos el primer y tercer cuartil (Q1 y Q3, respectivamente), de cada atributo.

```
[ ]: Q1 = []
Q3 = []
for idx in range(NoAtributos):
    if str(type(DBar[0][idx]))[8 : -2] != 'str':
        atrib = DBar.T[idx].tolist()
        atrib.sort()

        NoCuartil1 = 0.25 * (NoInstancias + 1)
        if str(type(NoCuartil1))[8 : -2] != 'int':
            pos1 = round(NoCuartil1)
            if pos1 < NoCuartil1:
                pos2 = pos1 + 1
            else:
                pos2 = pos1 - 1
            NoCuartil1 = round((pos1 + pos2) / 2)
        Cuartil1 = atrib[NoCuartil1 + 1]
        incremento = 1
        while True:
            if str(Cuartil1) == 'nan':
                Cuartil1 = atrib[NoCuartil1]
                NoCuartil1 -= 1
            else:
                break

        NoCuartil3 = 0.7 * (NoInstancias + 1)
        if str(type(NoCuartil3))[8 : -2] != 'int':
            pos1 = round(NoCuartil3)
            if pos1 < NoCuartil3:
                pos2 = pos1 + 1
            else:
                pos2 = pos1 - 1
            NoCuartil3 = round((pos1 + pos2) / 2)
        Cuartil3 = atrib[NoCuartil3 + 1]
        while True:
```

```

        if str(Cuartil3) == 'nan':
            Cuartil3 = atrib[NoCuartil3]
            NoCuartil3 -= 1
        else:
            break

    Q1.append(Cuartil1)
    Q3.append(Cuartil3)
Q1 = np.array(Q1)
Q3 = np.array(Q3)

```

4.2.2 Detectamos outliers y eliminamos sus respectivas instancias.

Calculamos el rango intercuantílico de cada atributo.

```

[ ]: IQR = []
    for idx in range(len(Q1)):
        IQR.append(Q3[idx] - Q1[idx])

```

Buscamos valores atípicos y el atributo en el que se encuentra ese valor (Seleccionamos quitar solo outliers extremos con OutType = 3, o también leves con OutType = 1.5).

```

[ ]: OutType = 3

Inst2Elim = []
for idx, Atrib in enumerate(DBar.T):
    probe1 = Q1[idx] - OutType * IQR[idx]
    probe2 = Q3[idx] + OutType * IQR[idx]
    for idx2, Val in enumerate(Atrib):
        if ((Val < probe1) or (Val > probe2)) and (idx2 not in Inst2Elim):
            Inst2Elim.append(idx2)

```

Generamos una nuevo array de datos eliminando las instancias que contienen algún outlier.

```

[ ]: DBW0 = []
    for InstNum in range(len(DBar)):
        if InstNum not in Inst2Elim:
            DBW0.append(DBar[InstNum])

    DBW0ar = np.array(DBW0)

```

Desplegamos los valores sin outliers, los cuales fueron identificados considerando el rango intercuantílico.

```

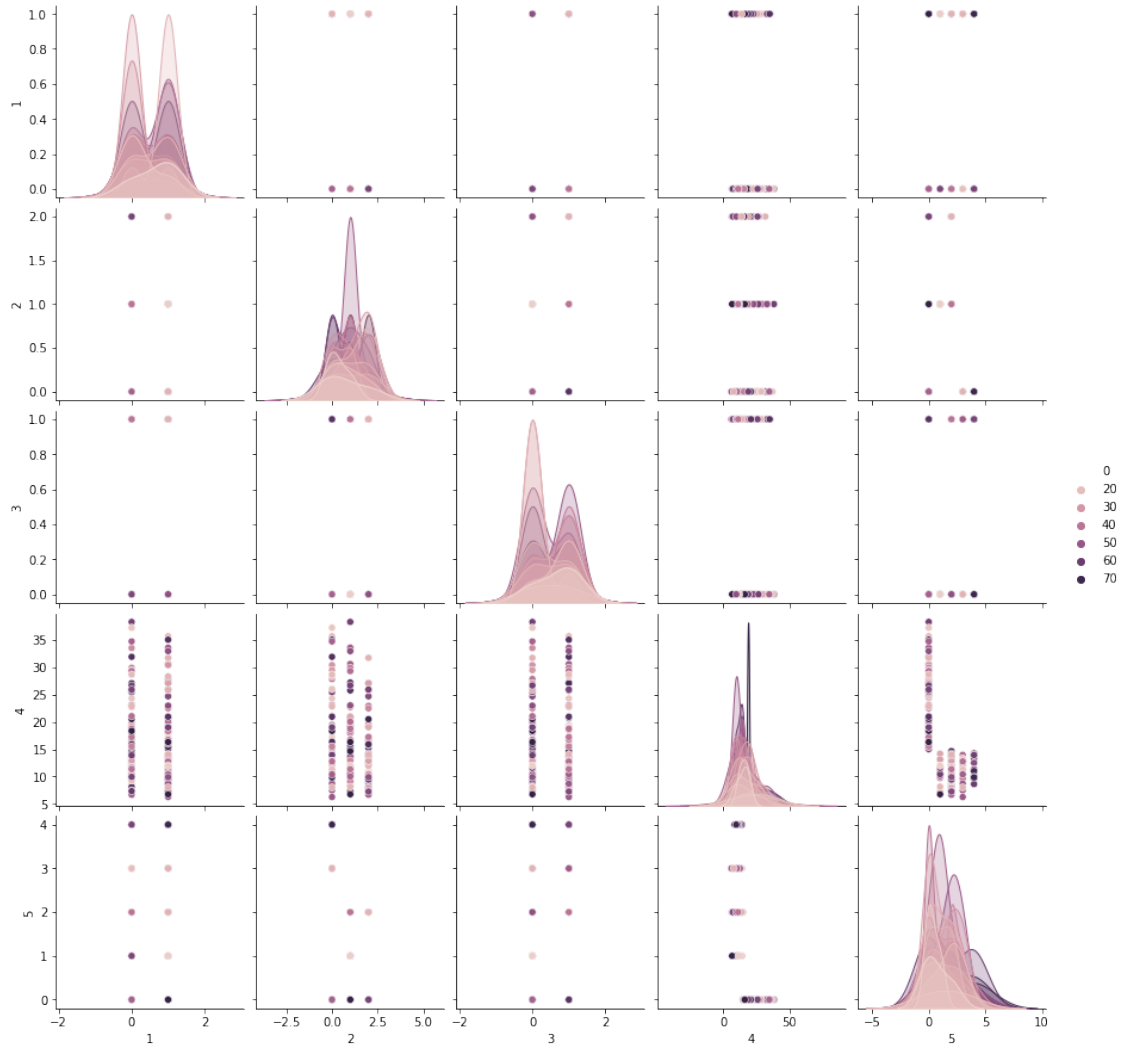
[ ]: showDB = pd.DataFrame(DBW0ar)
    sns.pairplot(showDB, vars = showDB.columns[1:6], hue = showDB.columns[0])

```

```

[ ]: <seaborn.axisgrid.PairGrid at 0x7ff8e66d4f90>

```



4.2.3 Normalizamos los datos

Elegimos un rango de normalización entre 0 y 1.

```
[ ]: MaximoNormalizado = 1
      MinimoNormalizado = 0
      RangoNormalizado = MaximoNormalizado - MinimoNormalizado
```

Normalizamos los valores y obtenemos el nuevo arreglo de valores normalizados 'DBNar'.

```
[ ]: DBNorm = []
      for idx in range(NoAtributos):

          CaractNorm = []
          if str(type(DBW0ar[0][idx]))[8 : -2] != 'str':
```

```

RangodeDatos = MaximoDeAtributos[idx] - MinimoDeAtributos[idx]
for idx2 in range(NoInstancias):

    if str(DBW0ar[idx2][idx]) != 'nan':
        D = DBW0ar[idx2][idx] - MinimoDeAtributos[idx]
        DPct = D / RangodeDatos
        dNorm = RangoNormalizado * DPct
        Normalizado = MinimoNormalizado + dNorm
        CaractNorm.append(Normalizado)
    else:
        CaractNorm.append(DBW0ar[idx2][idx])

DBNorm.append(CaractNorm)

DBNar = np.array(DBNorm)

```

Visualizamos una parte de la base de datos con los valores normalizados, para garantizar una correcta transformación.

```

[ ]: DBNarT = DBNar.T
showDB = pd.DataFrame(DBNarT)
showDB.head()

```

```

[ ]:
      0      1      2      3      4      5
0  0.135593  0.0  0.0  0.0  0.596848  0.00
1  0.542373  1.0  0.5  0.0  0.213397  0.25
2  0.542373  1.0  0.5  0.0  0.120239  0.25
3  0.220339  0.0  1.0  0.0  0.047814  0.50
4  0.779661  0.0  0.5  0.0  0.368191  0.00

```

4.3 Análisis de los datos

4.3.1 Serie de funciones creadas para obtener las métricas de rendimiento de un modelo.

```

[ ]: #Función para obtener el error cuadrático medio, recibe las listas de los
      ↪ valores predichos y los valores reales.
def MSE(MSEpred, MSEreal):
    MSEpredN = []
    for MSEp in MSEpred:
        MSEpredN.append(MSEp/100)

    MSErealN = []

```

```

for MSEr in MSEreal:
    MSErealN.append(MSEr/100)

MSEsize = len(MSErealN)
MSEt = 0
for MSEidx in range(MSEsize):
    MSEt += (MSEpredN[MSEidx] - MSErealN[MSEidx])**2
MSEf = MSEt / MSEsize
return(MSEf)

#Función para calcular la tasa de clasificación, recibe las listas de los
→valores predichos y los valores reales.
def TC(TCpred, TCreall):
    TCsize = len(TCpred)
    TCt = 0
    for TCidx in range(TCsize):
        if TCpred[TCidx] != TCreall[TCidx]:
            TCt += 1
    TCf = 1 - TCt / TCsize
    return(TCf)

#Función que obtiene los valores de la matriz de confusión (TP, FN, FP, TN),
→recibe las listas de los valores predichos y los valores reales.
def calculo_error(confusion_matrix):
    FP = confusion_matrix.sum(axis=0) - np.diag(confusion_matrix)
    FN = confusion_matrix.sum(axis=1) - np.diag(confusion_matrix)
    TP = np.diag(confusion_matrix)
    TN = confusion_matrix.sum() - (FP + FN + TP)

    FP = min(FP)
    FN = min(FN)
    TP = min(TP)
    TN = min(TN)
    return (TP, FN, FP, TN)

#Funciones para calcular la tasa de clasificación (TdC), incluyendo la exactitud
→(TdCExact), precisión (TdCPre), sensibilidad o Recall (TdCSens)
#y puntaje F - beta con beta = 1, es decir, F1 (TdCF1).
def TdC(TdCTP, TdCFN, TdCFP, TdCTN):
    TdCExact = (TdCTP + TdCTN) / (TdCTP + TdCTN + TdCFP + TdCFN)
    TdCPre = TdCTP / (TdCTP + TdCFP)
    TdCSens = TdCTP / (TdCTP + TdCFN)
    TdCF1 = (2 * TdCTP) / (2 * TdCTP + TdCFP + TdCFN)
    return([TdCExact, TdCPre, TdCSens, TdCF1])

#Función para obtener las estadísticas de rendimiento, recibe las listas de los
→valores predichos y los valores reales.

```

```
def Estadisticas(EYPred, EYreal):
    EMSE = MSE(EYPred, EYreal)
    ETC = TC(EYPred, EYreal)
    CM = confusion_matrix(EYPred, EYreal)
    ETP, EFN, EFP, ETN = calculo_error(CM)
    EExact, EPrecis, ESens, EF1 = TdC(ETP, EFN, EFP, ETN)
    Estadistica = [EMSE, ETC, EExact, EPrecis, ESens, EF1]
    return(Estadistica)
```

4.3.2 Creación de una clase propia, la cual fungirá como un árbol de decisión usando el algoritmo de ID3.

```
[ ]: class DecisionTree_ID3():
    def Entropia(self, YE):
        YE = list(YE)
        TotalE = len(YE)
        TiposE = np.unique(YE)
        ET = 0
        for TipoE in TiposE:
            PEi = YE.count(TipoE) / TotalE
            ET -= PEi * mt.log2(PEi)
        return(ET)

    def Ganancia(self, EG, AtributoG, YG):
        AtributoGarr = AtributoG
        AtributoG = list(AtributoG)
        TotalG = len(AtributoG)
        TiposG = np.unique(AtributoG)
        GT = EG
        for TipoG in TiposG:
            posG = np.where(AtributoGarr == TipoG)[0]
            posG = np.unique(posG)
            YGSel = []
            for pG in posG:
                YGSel.append(YG[pG])
            PGi = AtributoG.count(TipoG) / TotalG
            GT -= PGi * self.Entropia(YGSel)
        return(GT)

    def PRN(self, XRN, YRN):
        SRN = self.Entropia(YRN)
        GananciasRN = []
        XRN = np.array(XRN)
        for AtributoRN in XRN.T:
            GananciasRN.append(self.Ganancia(SRN, AtributoRN, YRN))
        Rmax = max(GananciasRN)
        GananciasRN = np.array(GananciasRN)
```



```

PosR = np.where(GananciasRN == Rmax)[0][0]
return(PosR)

def FDataNRes(self, XFDNR, YFDNR, RootPosFDNR, RootFDNR, LeafsFDNR):#, YGFR):
    XFDNRarr = np.array(XFDNR)
    XnewFDNR = []
    for idxFDNR, AtribFDNR in enumerate(XFDNRarr.T):
        if idxFDNR != RootPosFDNR:
            AtribFDNR = list(AtribFDNR)
            XnewFDNR.append(AtribFDNR)
    XnewFDNR = np.array(XnewFDNR).T
    XnewFDNR = XnewFDNR.tolist()

    XOutFDNR = []
    YOutFDNR = []
    if len(XnewFDNR) != 0:
        for LeafFDNR in LeafsFDNR:
            PosLeaf = np.where(RootFDNR == LeafFDNR)[0]
            X2Add = []
            Y2Add = []
            for idxFDNR in PosLeaf:
                X2Add.append(XnewFDNR[idxFDNR])
                Y2Add.append(YFDNR[idxFDNR])
            XOutFDNR.append(X2Add)
            YOutFDNR.append(Y2Add)

    LeafResFDNR = []
    LeafStatusFDNR = []
    for YLeafFDNR in YOutFDNR:
        CounterFDNR = Counter(YLeafFDNR)
        FirstFR = CounterFDNR.most_common(1)
        LeafResFDNR.append(FirstFR[0][0])

        if len(np.unique(YLeafFDNR)) == 1:
            LeafStatusFDNR.append(1)
        else:
            LeafStatusFDNR.append(0)
    return(XOutFDNR, YOutFDNR, LeafResFDNR, LeafStatusFDNR)

def FindRoot(self, XFR, YFR):
    RootPosFR = self.PRN(XFR, YFR)
    XFRarr = np.array(XFR)
    Root = XFRarr.T[RootPosFR]
    LeafsFR = np.unique(Root)
    DataNRes = self.FDataNRes(XFR, YFR, RootPosFR, Root, LeafsFR)#, YGFR)
    Data4LeafXFR = DataNRes[0]
    Data4LeafYFR = DataNRes[1]

```

```

LeafsResultsFR = DataNRes[2]
LeafsStatusFR = DataNRes[3]
return(RootPosFR, LeafsFR, Data4LeafXFR, Data4LeafYFR, LeafsResultsFR,
↳ LeafsStatusFR)

def PredDato(self, XPD, DTPD):
    ProfundidadPD = 0
    RootPD = XPD[DTPD[0][ProfundidadPD][0]]
    LeafsPD = DTPD[1][ProfundidadPD]
    NextPosPD = np.where(LeafsPD == RootPD)[0][0]
    Ypred = DTPD[4][ProfundidadPD][NextPosPD]
    while (DTPD[5][ProfundidadPD][NextPosPD] != 1):
        try:
            ProfundidadPD += 1
            RootPD = XPD[DTPD[0][ProfundidadPD][NextPosPD]]
            LeafsPD = DTPD[1][ProfundidadPD][NextPosPD]
            NextPosPD = np.where(LeafsPD == RootPD)[0][0]
            Ypred = DTPD[4][ProfundidadPD][NextPosPD]
        except:
            break
    return(Ypred)

def fit(self, Xtrainfit, Ytrainfit):
    AtribQuantity = len(Xtrainfit.T)
    RootRes = self.FindRoot(Xtrainfit, Ytrainfit)
    RootPos = RootRes[0]
    Leafs = RootRes[1]
    Data4LeafX = RootRes[2]
    Data4LeafY = RootRes[3]
    LeafsResults = RootRes[4]
    LeafsStatus = RootRes[5]
    FinalStatus = len(np.unique(LeafsStatus))
    UsedAtrib = 1
    self.Profundidad = 0
    self.DT = [[RootPos], [Leafs], [Data4LeafX], [Data4LeafY], [LeafsResults],
↳ [LeafsStatus]]
    while (UsedAtrib != AtribQuantity) and (FinalStatus != 1):
        self.Profundidad += 1
        self.DT[0].append([])
        for idx, Root in enumerate(self.DT[1][self.Profundidad-1]):
            Xtemp = self.DT[2][self.Profundidad-1][idx]
            Ytemp = self.DT[3][self.Profundidad-1][idx]
            if self.DT[5][self.Profundidad-1][idx] != 1:
                RootRes = self.FindRoot(Xtemp, Ytemp)
                RootPos = RootRes[0]
                Leafs = RootRes[1]
                Data4LeafX = RootRes[2]

```

```

        Data4LeafY = RootRes[3]
        LeafsResults = RootRes[4]
        LeafsStatus = RootRes[5]
        self.DT[0][self.Profundidad].append(RootPos)
        self.DT[1].append(Leafs)
        self.DT[2].append(Data4LeafX)
        self.DT[3].append(Data4LeafY)
        self.DT[4].append(LeafsResults)
        self.DT[5].append(LeafsStatus)
        UsedAtrib += 1
        FinalStatus = len(np.unique(self.DT[5][self.Profundidad]))

def GetPred(self, Datos):
    Ypred = []
    for data in Datos:
        Ypred.append(self.PredDato(data, self.DT))
    return(Ypred)

```

4.4 Entrenamos un modelo con el algoritmo de KNN generado.

4.4.1 Generación de X e Y

Agregamos aleatoriedad al orden de la base de datos.

```
[ ]: RDBNar = rd.sample(DBNar.T.tolist(), k=len(DBNar.T))
      RDBNar = np.array(RDBNar).T
```

Seleccionamos el atributo para nuestro valor objetivo 'Y' y el resto 'X' como ejemplos para el entrenamiento.

```
[ ]: X = DBNar[1:-2]
      X = X.T
      Y = DBNar[-1]
```

4.4.2 Entrenamos el modelo

Haremos 5 pruebas, eligiendo K=5 para el método de K-FOLD, en este caso, 5-FOLD.

La base de datos ya se encuentra ordenada aleatoriamente, por lo tanto solo tomaremos distintas secciones para el 20% de pruebas y su respectivo 80% para entrenamiento.

```
[ ]: Xres = X
      Yres = Y

      DBsize = len(Xres)
      DBp = DBsize / 5
      VDB = int(DBp)
      CDB = int(DBp * 2)
      SDB = int(DBp * 3)
      ODB = int(DBp * 4)

      Xtest1, Xtrain1, Ytest1, Ytrain1 = train_test_split(Xres, Yres, test_size = 0.
          ↪80, shuffle = False)

      Xtrain2, Ytrain2 = Xres[:VDB].tolist() + Xres[CDB:].tolist(), Yres[:VDB].
          ↪tolist() + Yres[CDB:].tolist()
      Xtrain2, Ytrain2 = np.array(Xtrain2), np.array(Ytrain2)
      Xtest2, Ytest2 = Xres[VDB:CDB], Yres[VDB:CDB]

      Xtrain3, Ytrain3 = Xres[:CDB].tolist() + Xres[SDB:].tolist(), Yres[:CDB].
          ↪tolist() + Yres[SDB:].tolist()
      Xtrain3, Ytrain3 = np.array(Xtrain3), np.array(Ytrain3)
      Xtest3, Ytest3 = Xres[CDB:SDB], Yres[CDB:SDB]

      Xtrain4, Ytrain4 = Xres[:SDB].tolist() + Xres[ODB:].tolist(), Yres[:SDB].
          ↪tolist() + Yres[ODB:].tolist()
      Xtrain4, Ytrain4 = np.array(Xtrain4), np.array(Ytrain4)
      Xtest4, Ytest4 = Xres[SDB:ODB], Yres[SDB:ODB]
```

```
Xtrain5, Xtest5, Ytrain5, Ytest5 = train_test_split(Xres, Yres, test_size = 0.  
→20, shuffle = False)
```

Entrenamos con KNN cada uno de los 5 grupos de datos Nota: Es posible elegir qué métrica ('MSE', 'TdC', 'Exactitud', 'Precision', 'Recall' o 'F1') tomar como referencia, en este caso se elige 'Precision'.

```
[ ]: model1 = DecisionTree_ID3()  
model1.fit(Xtrain1, Ytrain1)  
  
model2 = DecisionTree_ID3()  
model2.fit(Xtrain2, Ytrain2)  
  
model3 = DecisionTree_ID3()  
model3.fit(Xtrain3, Ytrain3)  
  
model4 = DecisionTree_ID3()  
model4.fit(Xtrain4, Ytrain4)  
  
model5 = DecisionTree_ID3()  
model5.fit(Xtrain5, Ytrain5)
```

Determinamos las predicciones de cada modelo.

```
[ ]: Ymodel1Pred = model1.GetPred(Xtest1)  
TasCla1 = TC(Ymodel1Pred, Ytest1)  
  
Ymodel2Pred = model2.GetPred(Xtest2)  
TasCla2 = TC(Ymodel2Pred, Ytest2)  
  
Ymodel3Pred = model3.GetPred(Xtest3)  
TasCla3 = TC(Ymodel3Pred, Ytest3)  
  
Ymodel4Pred = model4.GetPred(Xtest4)  
TasCla4 = TC(Ymodel4Pred, Ytest4)  
  
Ymodel5Pred = model5.GetPred(Xtest5)  
TasCla5 = TC(Ymodel5Pred, Ytest5)  
  
StadisticFin = (TasCla1 + TasCla2 + TasCla3 + TasCla4 + TasCla5) / 5
```

```
[ ]: print('Tasa de clasificación de la prueba 1 =', str(TasCla1) + '.')
      print('Tasa de clasificación de la prueba 2 =', str(TasCla2) + '.')
      print('Tasa de clasificación de la prueba 3 =', str(TasCla3) + '.')
      print('Tasa de clasificación de la prueba 4 =', str(TasCla4) + '.')
      print('Tasa de clasificación de la prueba 5 =', str(TasCla5) + '.')
      print('Precisión final =', str(StadisticFin) + '.')
```

Tasa de clasificación de la prueba 1 = 0.65.
 Tasa de clasificación de la prueba 2 = 0.475.
 Tasa de clasificación de la prueba 3 = 0.55.
 Tasa de clasificación de la prueba 4 = 0.375.
 Tasa de clasificación de la prueba 5 = 0.55.
 Precisión final = 0.5199999999999999.

4.5 Evaluamos el modelo

4.5.1 Analizamos métricas de evaluación

Tranformamos los datos de las etiquetas a enteros.

```
[ ]: Yt1 = []
      for Ytt1 in Ytest1:
          Yt1.append(int(Ytt1*100))

      Ytp1 = []
      for Yttp1 in Ymodel1Pred:
          Ytp1.append(int(Yttp1*100))

      Ytp2 = []
      for Yttp2 in Ymodel2Pred:
          Ytp2.append(int(Yttp2*100))

      Ytp3 = []
      for Yttp3 in Ymodel3Pred:
          Ytp3.append(int(Yttp3*100))

      Ytp4 = []
      for Yttp4 in Ymodel4Pred:
          Ytp4.append(int(Yttp4*100))

      Ytp5 = []
      for Yttp5 in Ymodel5Pred:
          Ytp5.append(int(Yttp5*100))
```

Generamos la tabla de métricas.

```
[ ]: from tabulate import tabulate

ModelStatistics = Estadisticas(Ytp1, Yt1)

STable = [ ['MSE', str(ModelStatistics[0])],
           ['Tasa de clasificación', str(ModelStatistics[1])],
           ['Exactitud', str(ModelStatistics[2])],
           ['Precisión', str(ModelStatistics[3])],
           ['Recall', str(ModelStatistics[4])],
           ['F1', str(ModelStatistics[5])] ]

print(tabulate(STable, headers = ['Métrica modelo 1', 'Valor'],
    ↳tablefmt="presto"))

ModelStatistics = Estadisticas(Ytp2, Yt1)

STable = [ ['MSE', str(ModelStatistics[0])],
           ['Tasa de clasificación', str(ModelStatistics[1])],
           ['Exactitud', str(ModelStatistics[2])],
           ['Precisión', str(ModelStatistics[3])],
           ['Recall', str(ModelStatistics[4])],
           ['F1', str(ModelStatistics[5])] ]

print(tabulate(STable, headers = ['Métrica modelo 2', 'Valor'],
    ↳tablefmt="presto"))

ModelStatistics = Estadisticas(Ytp3, Yt1)

STable = [ ['MSE', str(ModelStatistics[0])],
           ['Tasa de clasificación', str(ModelStatistics[1])],
           ['Exactitud', str(ModelStatistics[2])],
           ['Precisión', str(ModelStatistics[3])],
           ['Recall', str(ModelStatistics[4])],
           ['F1', str(ModelStatistics[5])] ]

print(tabulate(STable, headers = ['Métrica modelo 3', 'Valor'],
    ↳tablefmt="presto"))

ModelStatistics = Estadisticas(Ytp4, Yt1)

STable = [ ['MSE', str(ModelStatistics[0])],
           ['Tasa de clasificación', str(ModelStatistics[1])],
           ['Exactitud', str(ModelStatistics[2])],
           ['Precisión', str(ModelStatistics[3])],
           ['Recall', str(ModelStatistics[4])],
```

```

['F1', str(ModelStatistics[5])] ]

print(tabulate(STable, headers = ['Métrica modelo 4', 'Valor'],
    ↳tablefmt="presto"))

ModelStatistics = Estadisticas(Ytp5, Yt1)

STable = [ ['MSE', str(ModelStatistics[0])],
    ['Tasa de clasificación', str(ModelStatistics[1])],
    ['Exactitud', str(ModelStatistics[2])],
    ['Precisión', str(ModelStatistics[3])],
    ['Recall', str(ModelStatistics[4])],
    ['F1', str(ModelStatistics[5])] ]

print(tabulate(STable, headers = ['Métrica modelo 5', 'Valor'],
    ↳tablefmt="presto"))

```

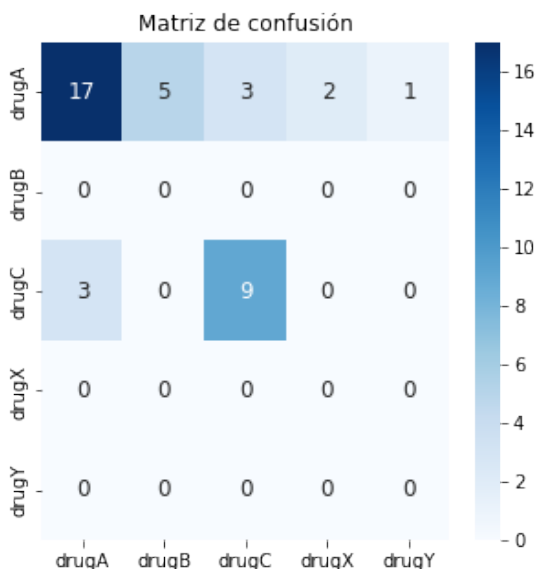
Métrica modelo 1		Valor	Exactitud		0.857143
-----+-----			Precisión		0
MSE		0.0984375	Recall		nan
Tasa de clasificación		0.65	F1		0
Exactitud		0.9	Métrica modelo 4		Valor
Precisión		0	-----+-----		
Recall		nan	MSE		0.154688
F1		0	Tasa de clasificación		0.425
Métrica modelo 2		Valor	Exactitud		0.833333
-----+-----			Precisión		0
MSE		0.135937	Recall		nan
Tasa de clasificación		0.425	F1		0
Exactitud		0.875	Métrica modelo 5		Valor
Precisión		0	-----+-----		
Recall		nan	MSE		0.142187
F1		0	Tasa de clasificación		0.425
Métrica modelo 3		Valor	Exactitud		0.875
-----+-----			Precisión		0
MSE		0.123438	Recall		nan
Tasa de clasificación		0.425	F1		0

Obtenemos la matriz de confusión de los datos de validación.

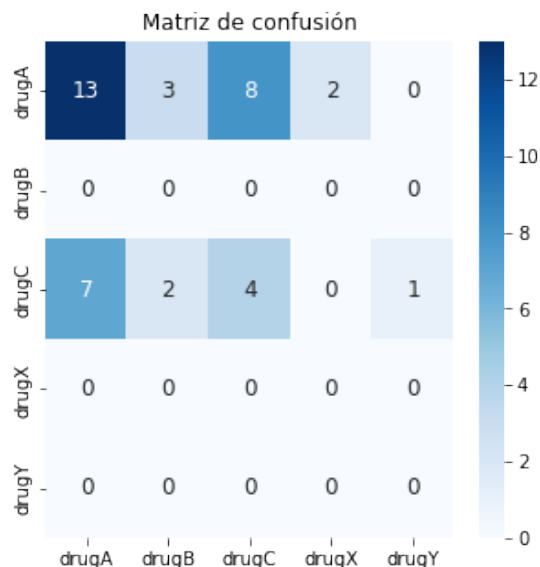
```
[ ]: def confusionMatrix(y_data_test, y_pred_test, titleChart, columns):
    cm = confusion_matrix(y_data_test, y_pred_test)
    cm=pd.DataFrame(cm, index=columns, columns=columns)
    plt.figure(figsize = (5,5))
    plt.title(titleChart)
    sns.heatmap(cm, annot = True,fmt="d", cmap='Blues' ,annot_kws={"size": 12})
    plt.show()

print('Matriz de confusión de la prueba 1\n')
confusionMatrix(Ytp1, Yt1, 'Matriz de confusión', list(np.
    ↳unique(DB_STAY['Drug'])))
print('\n\nMatriz de confusión de la prueba 2\n')
confusionMatrix(Ytp2, Yt1, 'Matriz de confusión', list(np.
    ↳unique(DB_STAY['Drug'])))
print('\n\nMatriz de confusión de la prueba 3\n')
confusionMatrix(Ytp3, Yt1, 'Matriz de confusión', list(np.
    ↳unique(DB_STAY['Drug'])))
print('\n\nMatriz de confusión de la prueba 4\n')
confusionMatrix(Ytp4, Yt1, 'Matriz de confusión', list(np.
    ↳unique(DB_STAY['Drug'])))
print('\n\nMatriz de confusión de la prueba 5\n')
confusionMatrix(Ytp5, Yt1, 'Matriz de confusión', list(np.
    ↳unique(DB_STAY['Drug'])))
```

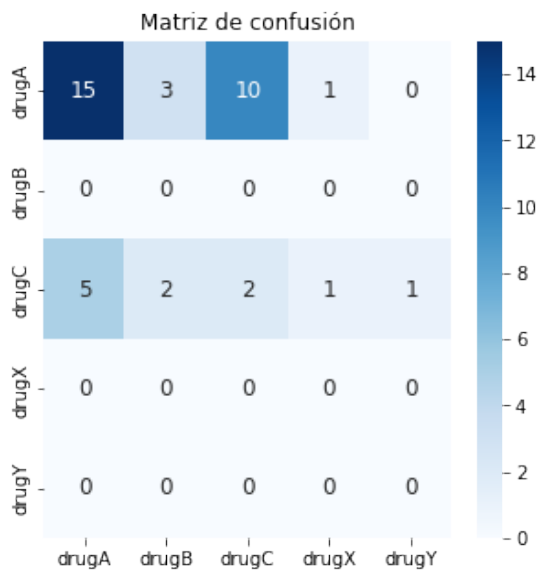
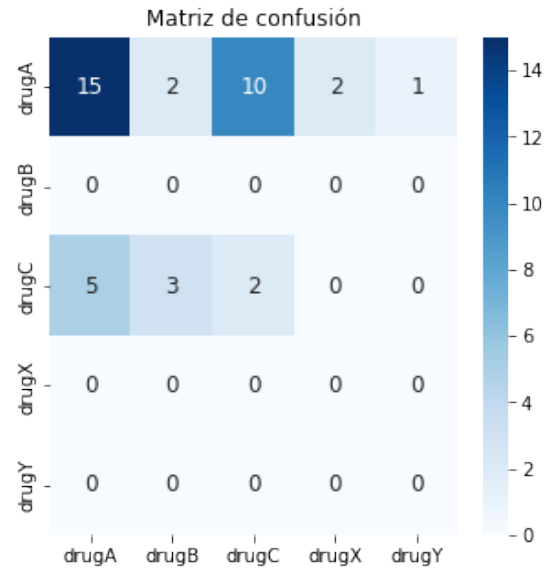
Matriz de confusión de la prueba 1



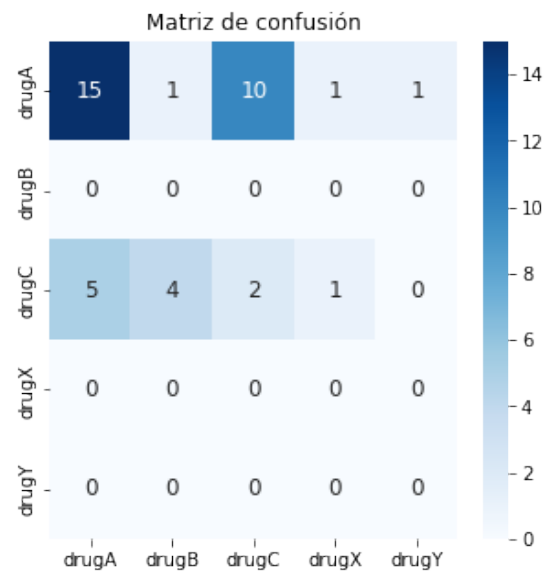
Matriz de confusión de la prueba 2



Matriz de confusión de la prueba 3



Matriz de confusión de la prueba 5



Finalmente, usaremos el intervalo de confianza Z para obtener los valores entre los que se encontraría la tasa de clasificación global.

```
[ ]: def media(Nums):
    n = len(Nums)
    Suma = 0
    for Num in Nums:
        Suma += Num
    ResM = Suma/n
    return(ResM)

def DesvEst(Nums):
    n = len(Nums)
    med = media(Nums)
    Suma = 0
    for Num in Nums:
        Suma += (Num - med)**2
    ResDE = (Suma / n)**0.5
    return(ResDE)

Cant_Pruebas = 50
TCs = []
for n in range(Cant_Pruebas):
    model = DecisionTree_ID3()
    model.fit(Xtrain1, Ytrain1)
    YmodelPred = model.GetPred(Xtest1)
    TasCla = TC(YmodelPred, Ytest1)
    TCs.append(TasCla)

MediaTC = media(TCs)
DesvEstTC = DesvEst(TCs)

Z = 1.96

RangoError = Z*(DesvEstTC/(Cant_Pruebas)**0.5)

print('Elegimos un valor de Z = 1.96 para un intervalo de confianza de 95%,
↳logrando así un valor global de tasa de clasifiacion =', str(round(MediaTC,
↳4)), ' y un intervalo de confianza de', str(RangoError) + '.')
print('\nLo anterior quiere decir, que el valor real de Tasa de clasificación
↳está entre', str(MediaTC - RangoError), 'y', str(MediaTC + RangoError) + '.')
```

Elegimos un valor de Z = 1.96 para un intervalo de confianza de 95%, logrando así un valor global de tasa de clasifiacion = 0.65 y un intervalo de confianza de 1.5386906095100993e-16.

Lo anterior quiere decir, que el valor real de Tasa de clasificación está entre 0.6499999999999994 y 0.6499999999999996.

5 Conclusión

Se ha implementado un algoritmo ID3 que sirve para realizar Árboles de Decisión, donde es posible analizar que es un algoritmo sencillo que puede implementarse de diferentes maneras, sin embargo, presenta de una forma sencilla, un algoritmo de clasificación útil para atributos cualitativos.

En este caso en particular se trató de forzar los límites del algoritmo, agregando muy pocos atributos, detalle por el cual es utilizado en ocasiones, y varias clases, teniendo un resultado no satisfactorio de clasificación, generando predicciones cercanas al 0.5, lo cual se puede tomar como sinónimo de no hacer una clasificación si no resultar en valores positivos por cuestiones de probabilidad.

Además, se refuerza la veracidad de los resultados con un análisis de su intervalo de confianza, el cual resulta básicamente en el mismo valor que las obtenidas en cada modelo individual.

6 Referencias

- [1] 1.10. Decision Trees. (s. f.). scikit-learn. Recuperado 10 de junio de 2022, de <https://scikit-learn/stable/modules/tree.html>
- [2] Bhardwaj, R., & Vatta, S. (2013). Implementation of ID3 algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(6).
- [3] Gómez, P. M. (2021). Drugs A, B, C, X, Y for decision trees [Data set].
- [4] Intervalo de Confianza. (s. f.). Disfrutalasmaticas.com. Recuperado 10 de junio de 2022, de <http://www.disfrutalasmaticas.com/datos/intervalo-confianza.html>
- [5] M. A. Aceves Fernández, *Inteligencia Artificial para programadores con prisa*. UNIVERSO de LETRAS, 2021.
- [6] Quinlan, J. R. (1996). Learning decision tree classifiers. *ACM Computing Surveys (CSUR)*, 28(1), 71-72.