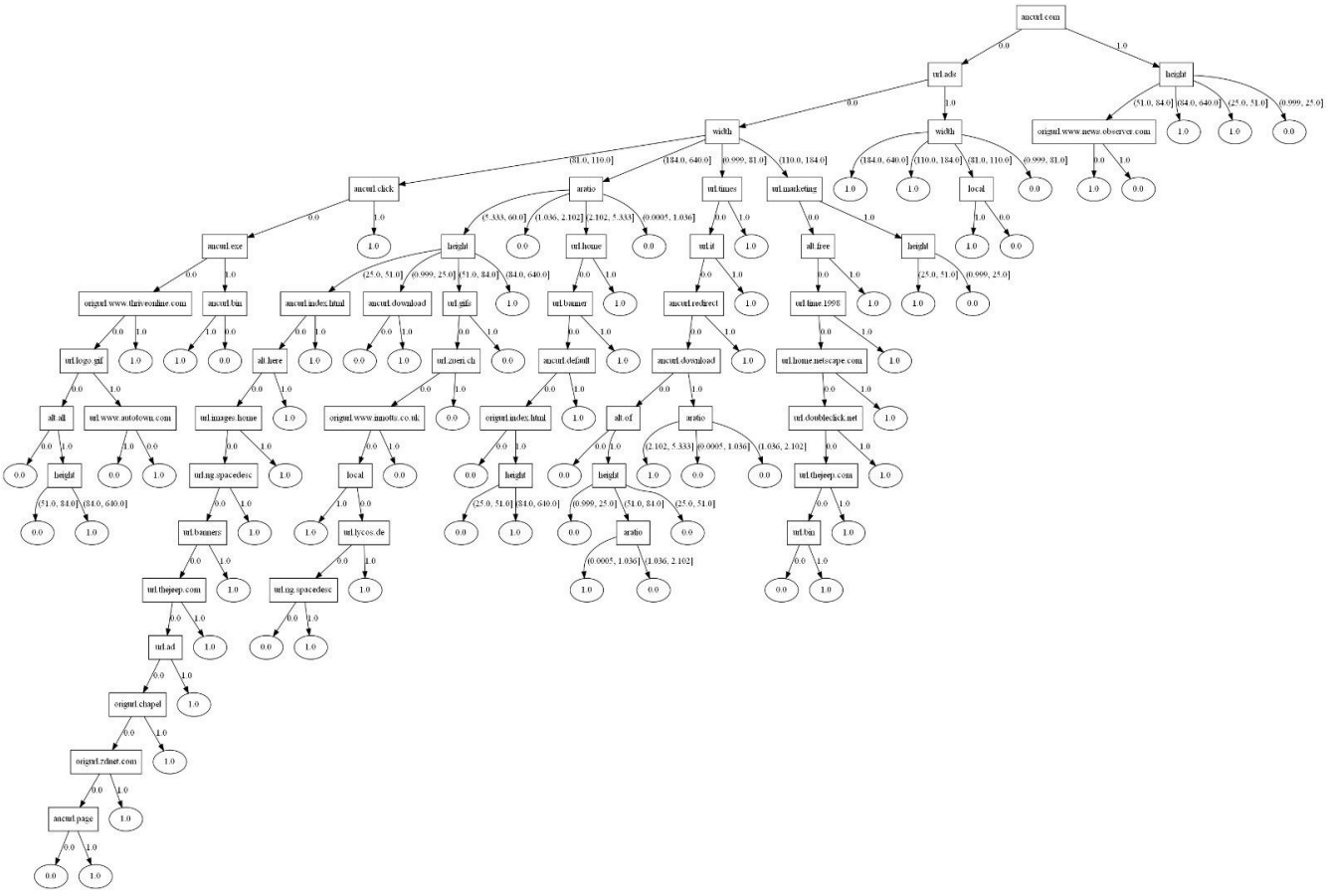
Memoria de la Práctica 2

Arboles de Decisión

Grupo: G8:30_3



Jorge Giménez Pérez – 1493035

Juan Carlos Soriano Valle – 1493037

Índice del contenido de la memoria

Introducción: Explicación del problema

El problema a resolver es realizar una correcta implementación de algoritmos que generen en Árboles de Decisión para una correcta predicción de un problema en concreto. Los algoritmos a tratar en esta práctica son los relativos a la ganancia de información, entropía, discriminación de conjuntos grandes de datos, etc.

Otro objetivo que se presenta en este problema se refiere a mejorar la compresión de los algoritmos de inferencia tales como ID3 i C4.5, hasta la posterior evaluación del algoritmo.

Dataset

El *Dataset* con el que se hará la práctica se llama *Internet Advertisements Data Set.* Este conjunto de datos sacado del repositorio especializado en *Machine Learning* trata de posibles anuncios en páginas web.

Data Set Characteristics:	Multivariate	**Number of Instances:**	3279	**Area:**	Computer
Attribute Characteristics:	Categorical, Integer, Real	**Number of Attributes:**	1558	**Date Donated**	1998-07-01
Associated Tasks:	Classification	**Missing Values?**	Yes	**Number of Web Hits:**	363522

Como podemos ver en la tabla de arriba, el *Dataset* está formado de 3279 instancias diferentes, donde cada una tiene un total de 1558 atributos, de clase categórica, entero y real. Esto nos genera un conjunto de datos enorme, por lo que los diferentes algoritmos propuestos como sus respectivas optimizaciones tienen que ser clave para obtener una clasificación con respecto el objetivo del estudio de este *Dataset*; construir un predictor que nos ayude a reconocer si un objetivo es un anuncio o no.

Si se observan los atributos de cada instancia, se puede observar como estos indican propiedades tales como la geometría de la imagen si la hubiera, las frases dentro de la *URL* observada, la *URL* de la imagen y su texto alternativo, la anchura del texto, etc.

De las 3279 instancias del conjunto, 2821 están catalogadas como NO anuncios y 458 como anuncios. Gracias a la futura separación entre datos *train* y datos *test*, esto nos servirá para probar cuan efectivo son nuestros algoritmos.

En los atributos nos encontramos 3 continuos. De estos datos, en un 28% de las instancias de nuestro set de datos nos encontramos que mínimo un atributo continuo falta, con lo cual tenemos que interpretarlo como desconocido y descartarlo o aplicar algún tipo de transformación (media, moda, etc.) con tal de normalizarlo con los datos y poder aplicar los diferentes algoritmos.

Solución Propuesta

Procesamiento de Datos

El *Dataset* propuesto viene en formato .csv. Esta clase de archivos permite tener todas las instancias con sus diferentes atributos separados por comas en un archivo con un peso mínimo.

El programa está codificado en *Python*, con distintos módulos donde se albergan las funciones utilizadas en el proceso de creación de los Árboles de Decisión.

```
"""
Script que procesa los ficheros de datos para pdoer hacerlos usables.
También genera archivos de test y de train además de discretizar columnas y
tartar los 'missing-values'.
"""
import pandas as pd
from modules.misc import deleteRowsWithValues, discretizeDataframe, train_test_split, getKfoldSubsets

dataset = pd.read_csv('./data/processed_advertisments_dataset.csv')
for contCol in ['height', 'width', 'aratio']:
    dataset.loc[dataset[contCol] == 0, contCol] = 'unknown'
dataset.loc[dataset['class'] == 'ad', 'class'] = 1
dataset.loc[dataset['class'] == 'noad', 'class'] = 0
dataset.to_csv('./data/finalAdvertismentsDataset.csv', index=False)

data = pd.read_csv('./data/finalAdvertismentsDataset.csv')
continuousColumns = ['height', 'width', 'aratio']
nbins = 4
data = deleteRowsWithValues(data, 'unknown')
data = discretizeDataframe(data, continuousColumns, nbins)

trainData, testData = train_test_split(data, 0.35)
trainData.to_csv('./data/train.csv', index=False)
testData.to_csv('./data/test.csv', index=False)

subsets = getKfoldSubsets(data, 5)
for partition in range(len(subsets)):
    subsets[partition].to_csv(f'./data/kfold/partition{partition+1}of{len(subsets)}.csv', index=False)
```

Archivo *processDatasetFiles.py*

En primer lugar, se tiene que operar con los archivos .csv en el que el *Dataset* está representado para que los algoritmos puedan tratar con los atributos de las distintas instancias. Para ello, nuestro archivo ***processDatasetFiles.py*** contiene todos los métodos necesarios para cumplir este objetivo.

Lo que hace este algoritmo es procesar los datos de manera que cualquier dato continuo que no es correcto lo establece en *unknown* de cara a un tratamiento posterior, ya que si no se modificase los resultados se verían afectados.

Se genera un archivo .csv alternativo donde estos datos fallidos son “arreglados” de cara a generar unos sets de *train/test* limpios. En este caso hacemos una separación del 35%.

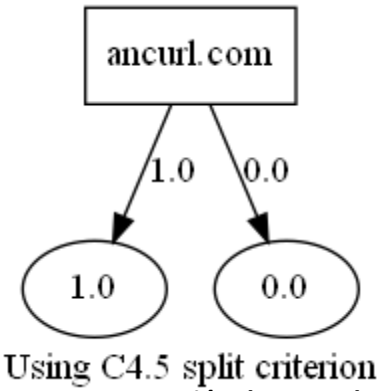
Por último, generamos 5 particiones para el *K-fold Validation* con el que evaluaremos los resultados finales de nuestro modelo predictor. De este último apartado hablaremos más tarde, cuando tratemos sobre la generación de los modelos de Árboles de Decisión.

Generación de Modelos

Clase DecisionTreeNode

Esta clase es la que se encarga de generar la estructura necesaria para obtener el Árbol de Decisión final. Con esta clase se implementan todos los nodos que pertenecen al Arbol de Decisión a construir.

Cada Nodo creado cuenta con atributos tales como el *Decision Tree* al cual pertenecen, el *Dataframe* de donde se han extraído los datos, si es un Nodo raíz o un Nodo hoja, etc.



Representación de un nodo del *DecisionTree*

```python
class DecisionTreeNode():
    """
    Clase que implementa cada uno de los nodos de los árboles de decisión.
    """
    def __init__(self, decisionTree = None, dataFrame = None, parentNode = None, splittingValue = None, isRoot=False, isLeaf=False):
        self.uuid = str(uuid.uuid4())
        self.decisionTree = decisionTree
        self.dataFrame = dataFrame
        self.parentNode = parentNode
        self.splittingValue = splittingValue
        self.isRoot = isRoot

        self.isLeaf = isLeaf
        self.splittingAttribute = None
        if isLeaf:
            self.prediction = self.dataFrame[self.decisionTree.targetAttribute].value_counts().index[0]
        else:
            self.prediction = None

        if decisionTree == None or decisionTree.splittingAlgorithm == None:
            self.splittingAlgorithm = None
        elif decisionTree.splittingAlgorithm == "ID3":
            self.splittingAlgorithm = ID3SplittingAlgorithm(self.dataFrame, self.decisionTree.targetAttribute, self.decisionTree.trueLabel)
        elif decisionTree.splittingAlgorithm == "C4.5":
            self.splittingAlgorithm = C4_5SplittingAlgorithm(self.dataFrame, self.decisionTree.targetAttribute, self.decisionTree.trueLabel)
        elif decisionTree.splittingAlgorithm == 'Gini' or 'gini' or 'GINI':
            self.splittingAlgorithm = GiniSplittingAlgorithm(self.dataFrame, self.decisionTree.targetAttribute, self.decisionTree.trueLabel)

        self.childrenNodes = []
        try:
            self.depth = 1 if self.isRoot else self.parentNode.depth+1
        except:
            self.depth = None
```

Clase *DecisionTreeNode*

Esta clase se comporta diferente según el algoritmo separador que se haya escogido en el conjunto del árbol, contando con 3 alternativas: *ID3, C4.5 o Gini.* Dependiendo de este atributo el Nodo será tratado con un algoritmo diferente.

Al tener un *Dataset* demasiado grande, hemos implementado un algoritmo que nos guarda el estado actual del árbol que se está construyendo. Esto nos ayuda en gran medida a reducir el coste de nuestro programa ya que el hecho de generar un árbol de cierta profundidad tarda desde minutos hasta horas, dependiendo de la profundidad que se haya escogido.

```python
    def toJSON(self):
        """
        Método que convierte en JSON el nodo del árbol para poder guardarlo en fichero JSON y
        posteriormente cargarlo.
        """
        node = {}
        node['uuid'] = str(self.uuid)
        node['depth'] = self.depth
        if self.parentNode != None: node['parent_uuid'] = str(self.parentNode.uuid)
        if self.splittingValue != None: node['splittingValue'] = str(self.splittingValue)
        if self.splittingAttribute != None: node['splittingAttirbute'] = self.splittingAttribute
        node['children'] = []
        if self.isLeaf:
            node['isLeaf'] = True
            node['prediction'] = self.prediction
            return node
        else:
            node['isLeaf'] = False
            node['children'] = []
            for child in self.childrenNodes:
                node['children'].append(child.toJSON())
        return node
```

Función *toJSON*

Esta función nos crea un archivo *JSON* del nodo para guardar su estado actual con todos sus atributos para cargarlo posteriormente y no tener que generar el mismo árbol de nuevo. Esto nos ahorra una cantidad de tiempo muy grande teniendo en cuenta las dimensiones de los datos.

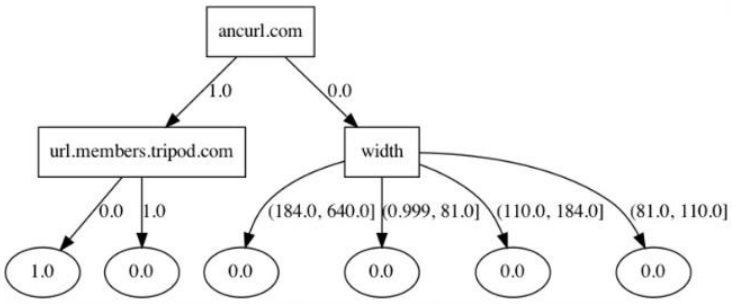
```
def fromDict(self, nodeDict, parentNode):
    """
    Método que construye un DecisionTreeNode desde un diccionario de python. (Se usa cuando
    se quieren generar los nodos de un árbol guardado en ficheor JSON)
    """
    self.uuid = nodeDict['uuid'] if nodeDict['uuid'] else None
    self.parentNode = parentNode
    self.depth = nodeDict['depth'] if nodeDict['depth'] else None
    self.splittingAttribute = nodeDict['splittingAttirbute'] if 'splittingAttirbute' in nodeDict else None
    self.splittingValue = nodeDict['splittingValue'] if 'splittingValue' in nodeDict else None
    self.isRoot = True if parentNode == None else False
    self.isLeaf = nodeDict['isLeaf']
    self.childrenNodes = []
    if not self.isLeaf:
        for childDict in nodeDict['children']:
            self.childrenNodes.append(DecisionTreeNode().fromDict(childDict, self))
    self.prediction = nodeDict['prediction'] if nodeDict["isLeaf"] else None
    return self



def getChildrenNodes(self):
    """
    Método que devuelve los DecisionTreeNode descendientes del actual. (Si el actual es un nodo hoja
    devuelve una lista vacía)
    """
    splittingAttribute = self.splittingAlgorithm.getSplittingAttribute()
    self.splittingAttribute = splittingAttribute
    possibleChoices = self.dataFrame[splittingAttribute].unique()
    childNodes = []
    for value in possibleChoices:
        subset = self.dataFrame[self.dataFrame[splittingAttribute]==value]
        subset.drop(splittingAttribute, axis=1, inplace=True)
        isLeaf = True if subset[self.decisionTree.targetAttribute].nunique() == 1 or self.depth == self.decisionTree.maxDepth else False
        childNodes.append(DecisionTreeNode(self.decisionTree, subset, self, value, isLeaf=isLeaf))
    return childNodes
```

Funciones encargadas de importar los archivos *JSON*

Cuando queremos importar estos archivos *JSON*, las funciones *fromDict* y *getChildrenNodes* se encargan de construir un *DecisionTreeNode* desde los ficheros *JSON.* Nuevamente, este proceso es fundamental por las características de nuestro *Dataset,* ya que si no se hiciese este proceso el programa subiría su tiempo de ejecución sensiblemente.

Clase DecisionTree

Esta clase se encarga de implementar los árboles de decisión i posteriormente visualizarlos. Dependiendo de los parámetros que se pasan al constructor, se pueden crear árboles de decisión usando los distintos criterios de separación estudiados: *ID3, C4.5 y Gini.*



DecisionTree

```python
class DecisionTree():
    """
    Esta es la clase que implementa los arboles de decisión i permite visualizarlos.
    Dependiendo de los parámetros que se pasen al constructor, se pueden crear árboles
    de decisión usando dinstintos criterios de separación (ID3, C4.5 y Gini).
    """
    def __init__(self, dataFrame = None, targetAttribute = None, trueLabel = None, falseLabel = None, splittingAlgorithm = None, maxDepth = None):
        """
        dataFrame: pandas.DataFrame sobre el que queremos generar el árbol de decisión.
        targetAttribute: String que coincide con el nombre de la columna del dataFrame que queremos usar como objetivo del árbol de decisión.
        trueLabel: String/Número/Booleano que indica de que manera representa la columna objetivo el valor positivo, por ejemplo: "Si", 1, True...
        splittingAlgorithm: String que indica que criterio de separación queremos usar: "ID3", "C4.5", "Gini".
        maxDepth: Número entero que determina la profundidad máxima del árbol.
        """
        self.dataFrame = dataFrame
        self.splittingAlgorithm = splittingAlgorithm
        self.targetAttribute = targetAttribute
        self.trueLabel = trueLabel
        self.falseLabel = falseLabel
        self.maxDepth = maxDepth
        self.rootNode = DecisionTreeNode(self, dataFrame, None, None, isRoot=True)
```

Clase *DecisionTree*

Dentro de esta clase encontramos atributos tales como el *Dataframe* de donde se obtienen los datos, el algoritmo de separación que se ha escogido, la máxima profundidad del árbol, el nombre del atributo del *Dataframe* que queremos utilizar como objetivo del *Decision Tree*, etc.

```python
def generate(self):
    """
    Método que inicia la recursión para generar el árbol de decisión.
    """
    self.__build(self.rootNode)

def __build(self, currentNode):
    """
    Método (privado) recursivo que genera todos los nodos del árbol de decisión.
    """
    if currentNode.isLeaf: return
    childrenNodes = currentNode.getChildrenNodes()
    currentNode.childrenNodes = childrenNodes
    for childNode in childrenNodes:
        self.__build(childNode)
    return
```

Funciones para generar el árbol

```python
def predict(self, dataFrame):
    """
    Método que recive por parámetro un dataframe con un número indeterminado de registros
    y devuelve una matriz con cada una de las predicciones del árbol para esos registros.
    """
    predictions = []
    for row in range(dataFrame.shape[0]):
        register = dataFrame.iloc[row, :]
        actualNode = self.rootNode
        while not actualNode.isLeaf:
            match = False
            registerValue = register[actualNode.splittingAttribute]
            for nextNode in actualNode.childrenNodes:
                if nextNode.splittingValue == str(registerValue):
                    actualNode = nextNode
                    match = True
                    break
            if not match: actualNode = actualNode.childrenNodes[0]
        predictions.append(actualNode.prediction)
    return predictions
```

Funciones para predecir

También encontramos los métodos que sirven para generar el árbol con todos sus nodos, vistos previamente en la clase *DecisionTreeNode*. Dentro de las funciones de esta clase también se incluye el método predictor. Este método recibe por parámetro el *Dataframe* con el que se está trabajando y genera una matriz con las predicciones del árbol.

Con tal de visualizar el árbol resultante, la función *visualize* genera un .PNG con el árbol generado y su estructura final. Este se llama con un *String* como parámetro que será el título de la imagen resultante del árbol de decisión.

```
def visualize(self, title):
    """
    Renderiza un PNG con la estructura final del árbol de decisión.
    title: Parámetro al que se le pasa un String para que se use como título de la imagen en
    la parte inferior del PNG y como nombre del archivo generado.
    """
    dot = Digraph(comment="Graphic representation of the resulting decision tree", format='png')
    dot.attr(label=title)
    self.__buildVisualization(dot, None, self.rootNode)
    dot.render(f'./outputs/graphOutputs/{title}.gv', view=True)

def __buildVisualization(self, dot, previousNode, currentNode):
    """
    Método (privado) recursivo que genera la visualización del árbol.
    """
    if currentNode.isRoot:
        dot.node(currentNode.uuid, str(currentNode.splittingAttribute), shape="box")
        for childNode in currentNode.childrenNodes:
            self.__buildVisualization(dot, currentNode, childNode)
    elif currentNode.isLeaf:
        dot.node(currentNode.uuid, str(currentNode.prediction), shape="ellipse")
        dot.edge(previousNode.uuid, currentNode.uuid, label=str(currentNode.splittingValue))
        return
    else:
        childrenNodes = currentNode.childrenNodes
        dot.node(currentNode.uuid, str(currentNode.splittingAttribute), shape="box")
        dot.edge(previousNode.uuid, currentNode.uuid, label=str(currentNode.splittingValue))
        for childNode in childrenNodes:
            self.__buildVisualization(dot, currentNode, childNode)
        return
```

Funciones para visualizar el árbol

Igual que en la clase *DecisionTreeNode*, esta clase también tiene una función para guardar todo el árbol en formato JSON. Esta función realiza una recursión que recorre todos los nodos llamando a la función *toJSON* alojada en la clase *DecisionTreeNode.* Lo mismo ocurre a la hora de importar, se abre el archivo *JSON* generado previamente y por cada nodo del árbol se ejecuta la función de la clase nodo para cargar el estado del nodo a partir del archivo que se está leyendo.

```
def saveToFile(self, fileName):
    """
    Método que inicia una recursión por cada nodo para convertirlo en formato JSON para poder
    guardar el árbol en un fichero de texto y poder cargarlo posteriormente.
    Guarda el JSON generado en la ruta ./outputs/modelsOutputs/<parámetro fileName>.json.
    """
    file = open(f'./outputs/modelsOutputs/{fileName}.json', 'w+')
    json.dump(self.rootNode.toJSON(), file)
    file.close()
```

Función para exportar el árbol a JSON

```
def fromJSON(self, path):
    file = open(path ,'r')
    treeDict = json.load(file)
    file.close()
    self.rootNode = DecisionTreeNode().fromDict(treeDict, None)
    return self
```

Función para importar el árbol desde JSON

Clase *SplittingAlgorithm*

En esta clase se implementan todos los algoritmos de separación que se usan en los distintos árboles de decisión para separar los datos. Se utilizan 3: *ID3, C4.5 y Gini.*

Los métodos generales que hay en esta clase son los de inicialización y el de obtener la entropía. En el primero, el de inicializar el algoritmo separador, se comienza describiendo que atributo del *Dataset* se escoge como objetivo, indicando a través del *trueLabel* que valor contienen las filas positivas de esta variable escogida como objetivo (*True, Si, 1, etc.*), para después calcular su entropía.

En la función de la entropía se ejecutan todos los cálculos necesarios para devolver tanto la entropía como las probabilidades respecto la columna que se ha seleccionado como objetivo.

```
def __init__(self, dataFrame, targetAttribute, trueLabel = 1):
    """
    Se inicializa con un dataframe de pandas i especificando el nombre de la columna objetivo.
    El parámetro "trueLabel" sirve para indicar qué valor contienen las filas positivas de
    la variable objetivo, normalmente '1' (contrario de '0') o 'True' (contrario de 'False').
    Por defecto se usa '1'.

    Seguidamente calcula la inicializa la entropía del conjunto.
    """
    self.dataFrame = dataFrame
    self.targetAttribute = targetAttribute
    self.trueLabel = trueLabel
    self.initialEntropy = self.getEntropy(self.dataFrame)

def getEntropy(self, subset):
    """
    Devuelve la entropía y las probabilidades respecto a la columna objetivo del dataframe
    que recibe por el parámetro subset.
    """
    totalCount = subset.shape[0]
    try:
        positiveCount = subset[subset[self.targetAttribute] == self.trueLabel].count()[1]
    except:
        positiveCount = 0
    negativeCount = totalCount - positiveCount
    positiveProbability = np.divide(positiveCount, totalCount)
    negativeProbability = np.divide(negativeCount, totalCount)

    positiveProbLog = 0 if positiveProbability == 0 else np.log2(positiveProbability)
    negativeProbLog = 0 if negativeProbability == 0 else np.log2(negativeProbability)
    positiveProbXLog = np.multiply(positiveProbability, positiveProbLog)
    negativeProbXLog = np.multiply(negativeProbability, negativeProbLog)
    entropy = -np.add(positiveProbXLog, negativeProbXLog)
    return entropy
```

Clase *SplittingAlgorithm*

Dentro de esta clase se encuentran las 3 funciones encargadas de la separación de los distintos nodos del árbol.

Estas funciones se representan en clases, donde cada clase es heredada de la clase base *SplittingAlgorithm*, teniendo así los métodos generales y los específicos de cada tipo de separación.

La clase *C4.5* incorpora el algoritmo que obtiene el atributo que obtiene el mejor ratio de ganancia y así separar los datos en relación al atributo seleccionado por el algoritmo.

De igual manera, en la clase *ID3*, heredada de la clase *SplittingAlgorithm,* encontramos el algoritmo que encuentra el atributo que tiene la mejor ganancia de información.

Por último, también hereda de la clase *SplittingAlgorithm* la clase *Gini*, que encuentra cual es el atributo que tiene el menor valor de *Gini* que facilita al algoritmo la eficiente separación de los datos respecto sus atributos.

```python
class ID3SplittingAlgorithm(SplittingAlgorithm):
    """
    Esta clase hereda de la clase base SplittingAlgorithm y es donde se implementa el
    algoritmo ID3 (Ganancia de información basado en entropía).
    """
    def __init__(self, dataFrame, targetAttribute, trueLabel):
        super().__init__(dataFrame, targetAttribute, trueLabel)

    def getSplittingAttribute(self):
        """
        Este método encuentra cuál es el atributo por el cuál se deben separar los datos para
        conseguir la mejor ganancia de información.
        """
        gains = []
        for column in self.dataFrame.columns[:-1]:
            entropies = []
            counts = []
            for value in self.dataFrame[column].unique():
                subset = self.dataFrame[self.dataFrame[column] == value]
                counts.append(subset.shape[0])
                entropy = self.getEntropy(subset)
                entropies.append(entropy)
            totalCount = np.sum(counts)
            informationGain = np.subtract(self.initialEntropy, np.sum(np.multiply(np.divide(counts, totalCount), entropies)))
            gains.append(informationGain)
        return self.dataFrame.columns[np.argmax(gains)]
```

Clase *ID3SplittingAlgorithm*

```python
class C4_5SplittingAlgorithm(SplittingAlgorithm):
    """
    Esta clase hereda de la clase base SplittingAlgorithm y es donde se implementa el
    algoritmo C4.5 (Proporción de la ganancia).
    """
    def __init__(self, dataFrame, targetAttribute, trueLabel):
        super().__init__(dataFrame, targetAttribute, trueLabel)

    def getSplittingAttribute(self):
        """
        Este método encuentra cuál es el atributo por el cuál se deben separar los datos para
        conseguir el mejor ratio de ganancia.
        """
        gainRatios = []
        for column in self.dataFrame.columns[:-1]: #['Wind', 'Outlook']:
            entropies = []
            counts = []
            for value in self.dataFrame[column].unique():
                subset = self.dataFrame[self.dataFrame[column] == value]
                counts.append(subset.shape[0])
                entropy = self.getEntropy(subset)
                entropies.append(entropy)
            totalCount = np.sum(counts)
            gain = self.initialEntropy - np.sum(np.multiply(np.divide(counts, totalCount), entropies))
            splitInfo = -np.sum(np.multiply(np.divide(counts, totalCount), np.log2(np.divide(counts, totalCount))))
            gainRatio = gain/splitInfo if splitInfo != 0 else 0
            gainRatios.append(gainRatio)
        return self.dataFrame.columns[np.argmax(gainRatios)]
```

Clase *C4.5SplittingAlgorithm*

```python
class GiniSplittingAlgorithm(SplittingAlgorithm):
    """
    Esta clase hereda de la clase base SplittingAlgorithm y es donde se implementa el
    algoritmo Gini.
    """
    def __init__(self, dataFrame, targetAttribute, trueLabel):
        super().__init__(dataFrame, targetAttribute, trueLabel)

    def getSplittingAttribute(self):
        "Este método encuentra cuál es el atributo por el cuál se debe separar los datos"
        giniIndexes = []
        for column in self.dataFrame.columns[:-1]:
            giniSubindexes = []
            counts = []
            for value in self.dataFrame[column].unique():
                generalSubset = self.dataFrame[self.dataFrame[column] == value]
                counts.append(generalSubset.shape[0])
                positiveSubset = generalSubset[generalSubset[self.targetAttribute] == self.trueLabel]
                negativeSubset = generalSubset[generalSubset[self.targetAttribute] != self.trueLabel]
                giniSubindex = np.subtract(1, np.add(np.square(np.divide(positiveSubset.shape[0],
                generalSubset.shape[0])),np.square(np.divide(negativeSubset.shape[0], generalSubset.shape[0]))))
                giniSubindexes.append(giniSubindex)
            totalCount = np.sum(counts)
            giniIndex = np.sum(np.multiply(giniSubindexes, np.divide(counts, totalCount)))
            giniIndexes.append(giniIndex)
        return self.dataFrame.columns[np.argmin(giniIndexes)]
```

Clase *GiniSplittingAlgorithm*

Funciones útiles

A parte de los métodos incluidos en las diferentes clases de los modelos, también hemos implementado otras funciones útiles para el correcto y optimo funcionamiento de cada uno de los algoritmos a ejecutar.

DiscretizeDataframe

Esta función se encarga de separar los atributos continuos que pueda tener el *Dataset* que devuelve un *pandas.DataFrame* con las columnas discretizadas en tantos cuartiles como indique el parámetro *n_bins* de la función *pd.qcut*.

```python
for column in columns:
    dataframe[column] = pd.qcut(dataframe[column], q=n_bins)
return dataframe
```

Función *DiscretizeDataframe*

DeleteRowsWithValues

La función se encarga de eliminar las instancias del *Dataset* que contengan en alguno de sus atributos el valor que recibe como parámetro. Normalmente se utiliza para eliminar las instancias que contienen valores desconocidos (*value="unknown"*). Devuelve un *pd.DataFrame* con todas las correcciones añadidas.

```
for column in dataframe.columns:
    dataframe = dataframe[dataframe[column] != value]
    dataframe[column] = dataframe[column].astype("float64")
return dataframe
```

Función *DeleteRowsWithValues*

TrainTestSplit

La función por excelencia en el aprendizaje computacional. Se encarga de dividir nuestro *Dataset* en partes de *train* y partes de *test.* El porcentaje de división del *Dataset* se pasa como parámetro en la variable *testRatio.*

```
nInstances = dataframe.shape[0]
nTestInstances = int(round(nInstances*testRatio))
nTrainInstances = nInstances - nTestInstances
dataframe = dataframe.sample(n=nInstances, random_state=0).reset_index(drop=True)
train = dataframe.iloc[0:nTrainInstances, :]
test = dataframe.iloc[nTrainInstances:nTrainInstances+nTestInstances, :]
return train, test
```

Función *TrainTestSplit*

GetKfoldSubsets

Esta función es muy importante ya que nos genera las particiones de *Kfold* deseadas para luego hacer el *Kfold Validation.*

```
nInstances = dataframe.shape[0]
print(nInstances)
partitionInstances = int(round(nInstances/k))
if partitionInstances*k > nInstances: partitionInstances -= 1
dataframe = dataframe.sample(n=nInstances, random_state=0).reset_index(drop=True)
subsets = []
for i in range(k):
    subset = dataframe.iloc[partitionInstances*i:partitionInstances*i+partitionInstances, :]
    subsets.append(subset)
return subsets
```

Función *GetKfoldSubsets*

Generación de Árboles de Decisión

Esta clase reúne todos los métodos y funciones explicados hasta ahora para generar el resultado final (*Decision Tree*) y su validación.

```
trainData = pd.read_csv('./data/train.csv')
yColumnName = 'class'
trueValue = 1.0
falseValue = 0.0

"""
Creamos modelos con distintas profundidades máximas para poder decidir
qué hiperparámetro de profundiad máxima nos conviene.
"""
for model in ['ID3', 'Gini', 'C4.5']:
    for maxDepth in [3, 5, 7, 9, 11, 13, 15]:
        print(f'Progress: Model -> {model} with maximum depth: {maxDepth}')
        tree = DecisionTree(trainData, yColumnName, trueValue, falseValue, model, maxDepth=maxDepth)
        tree.generate()
        tree.saveToFile(f'{model}_maxDepth{maxDepth}')
    print('\n')
'''
```

Antigua función generadora de *Decision Trees*

En una versión anterior del algoritmo, se creaban distintos árboles con los diferentes métodos de separación de nodos. De cada método de separación (*ID3, C4.5 y Gini)* se generaban distintos árboles con diferentes profundidades máximas tales como 3, 5, 7, 9, 11, 13 y 15 y posteriormente guardarlos en archivos *JSON* con los métodos enseñados anteriormente.

A parte de generar demasiados árboles para su correcta validación, esto comportaba una complejidad computacional muy alta, y una ejecución completa del programa podía llevar varias decenas de horas en completarse.

Por esta razón, diseñamos otro algoritmo más óptimo de cara a reducir esta complejidad computacional tan alta.

El primer punto a tener en cuenta en este nuevo algoritmo es la profundidad máxima establecida en 15. Este valor ha sido escogido para mantener la relación complejidad-fiabilidad, puesto que al hacer 1 árbol a profundidad 15 no supone la misma carga que 7 árboles de profundidad variable, obteniendo una fiabilidad suficientemente alta como para considerar el resultado final como válido.

El siguiente aspecto importante es la implementación del *Kfold Validation* como método de validación de los modelos generados. Generamos 5 particiones de datos para evaluarlas con el *Kfold*. La 1 por ejemplo utiliza las otras 4 particiones que no se han seleccionado para entrenar y crea un modelo con esta combinación. Con la partición 2, entrena con las particiones 1, 3, 4 y 5 y genera un modelo y así sucesivamente.

Dentro de este proceso que involucra el método de validación de *Kfold,* hemos implementado la utilización de 3 *threads* distintos donde cada uno ejecuta en un hilo un algoritmo diferente de separación. Esto nos ayuda a reducir el tiempo de generación de los modelos y su posterior validación sensiblemente.

```python
MAX_DEPTH = 15

kfoldPartitions = []
for i in range (1, 6):
    kfoldPartitions.append(pd.read_csv(f'./data/kfold/partition{i}of5.csv'))

def generateModelAndSave(tree, fileName):
    tree.generate()
    tree.saveToFile(fileName)

for kIdx in range(1,6):
    print(f"PROGRESS ===> Current K index {kIdx}/5")
    trainPartitions = []
    for partitionIdx in range(5):
        if partitionIdx+1 != kIdx: trainPartitions.append(kfoldPartitions[partitionIdx])
    trainData = pd.concat(trainPartitions)

    tree1 = DecisionTree(trainData, 'class', 1.0, 0.0, 'ID3', maxDepth=MAX_DEPTH)
    tree2 = DecisionTree(trainData, 'class', 1.0, 0.0, 'Gini', maxDepth=MAX_DEPTH)
    tree3 = DecisionTree(trainData, 'class', 1.0, 0.0, 'C4.5', maxDepth=MAX_DEPTH)
    t1 = threading.Thread(target=generateModelAndSave, args=(tree1, f'ID3_maxDepth{MAX_DEPTH}__Partition{kIdx}of5_isOutOfTraining'))
    t2 = threading.Thread(target=generateModelAndSave, args=(tree2, f'Gini_maxDepth{MAX_DEPTH}__Partition{kIdx}of5_isOutOfTraining'))
    t3 = threading.Thread(target=generateModelAndSave, args=(tree3, f'C4.5_maxDepth{MAX_DEPTH}__Partition{kIdx}of5_isOutOfTraining'))
    t1.start()
    t2.start()
    t3.start()
    t1.join()
    print('--- PREOGRESS ===> ID3 FINISHED')
    t2.join()
    print('--- PREOGRESS ===> Gini FINISHED')
    t3.join()
    print('--- PREOGRESS ===> C4.5 FINISHED')
    print('\n')
```

Función generadora de *Decision Trees*

Al acabar los 3 *threads* el programa ha generado 3 modelos diferentes de árboles de decisión, cada uno basándose en un método diferente de separación de datos, todos con profundidad máxima 15. Todo este proceso se repite 5 veces, una por cada partición de datos que nos ha generado el método de *Kfolding Validation*.

Al tener todos los 15 modelos creados y en busca de comprobar su eficacia y precisión, juntamos los 5 modelos según el método que hemos utilizado para separar los datos y hacemos la media de los resultados, para así obtener un valor lo mas parecido a la realidad posible sobre la eficacia predictiva de nuestro modelo creado, utilizando los datos de test que hemos dividido al principio del programa.

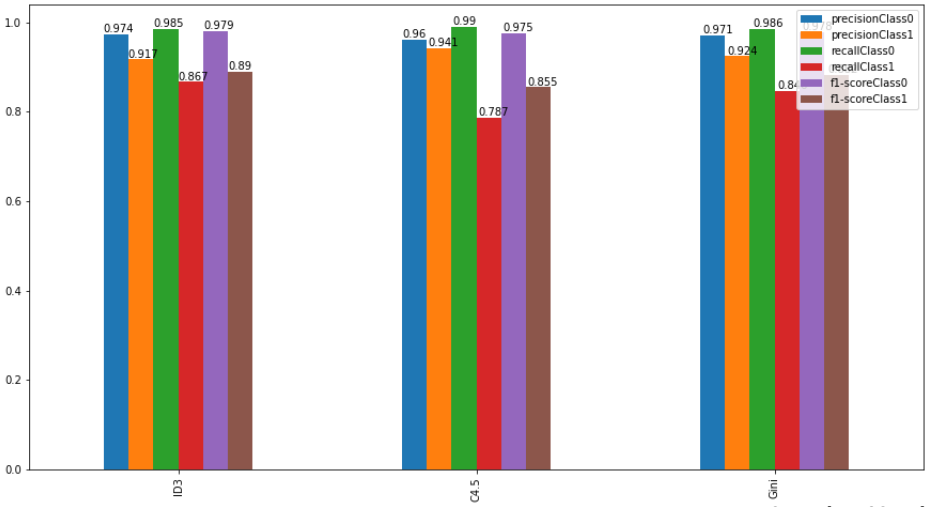
Resultados

Miramos los diferentes resultados que hemos obtenido al ejecutar el programa al completo. Tenemos resultados tanto de *ID3* como de *C4.5* como de *Gini,* obteniendo un árbol de decisión final de profundidad 15 muy robusto y completo.

	precisionClass0	precisionClass1	recallClass0	recallClass1	f1-scoreClass0	f1-scoreClass1
ID3	0.974	0.917	0.985	0.867	0.979	0.890
C4.5	0.960	0.941	0.990	0.787	0.975	0.855
Gini	0.971	0.924	0.986	0.846	0.978	0.882

Media de las evaluaciones de cada modelo en Kfold

Estos resultados corresponden a la evaluación de cada modelo usando la fracción del *Dataset* que no se utilizaron en su fase de *train* para luego hacer la media de los resultatos.

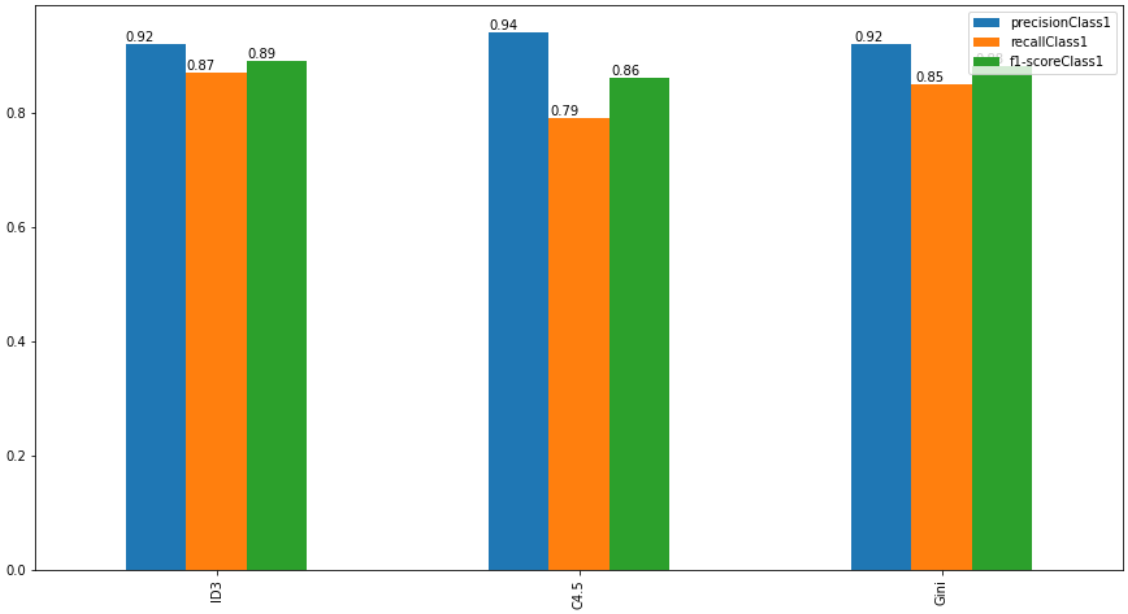


Media de las evaluaciones de cada modelo en Kfold (Gráfica)

Centrándonos en la clase 1.0 del *Dataframe* (‘ad’), dado que es la que tiene pocos registros y es la que nos interesa predecir mejor, obtenemos estos resultados:

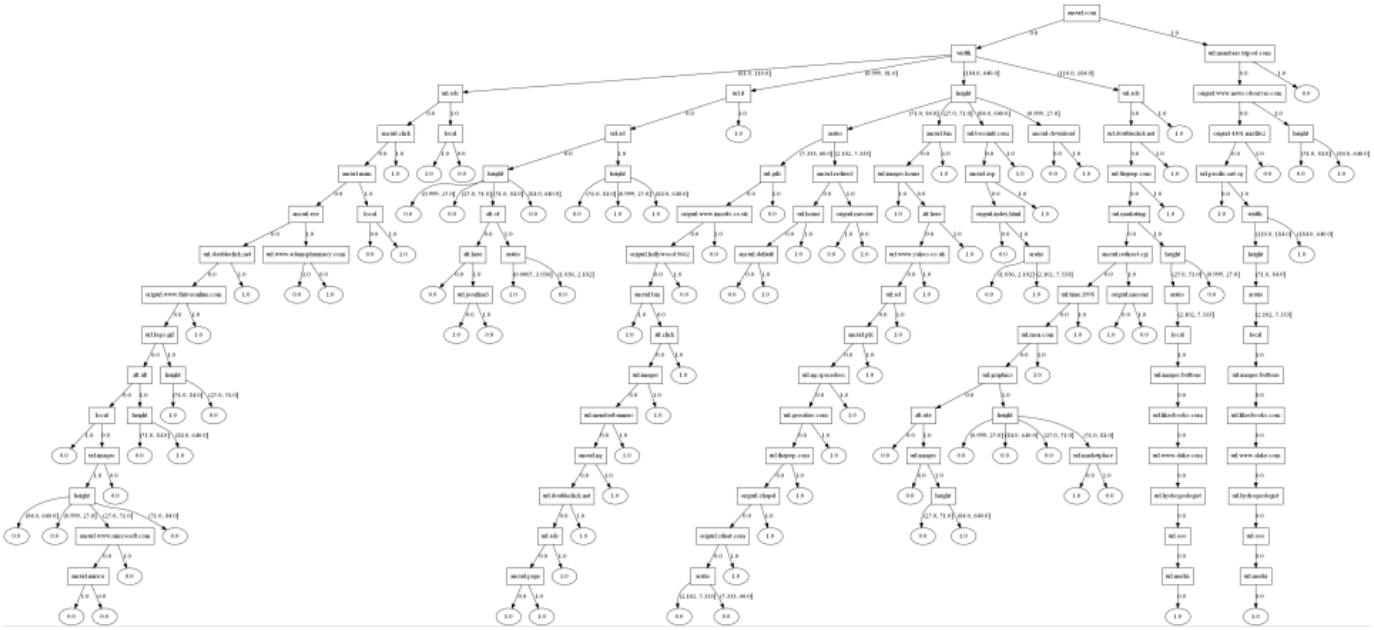
	precisionClass1	recallClass1	f1-scoreClass1
ID3	0.92	0.87	0.89
C4.5	0.94	0.79	0.86
Gini	0.92	0.85	0.88

Media de las evaluaciones de la clase 1.0

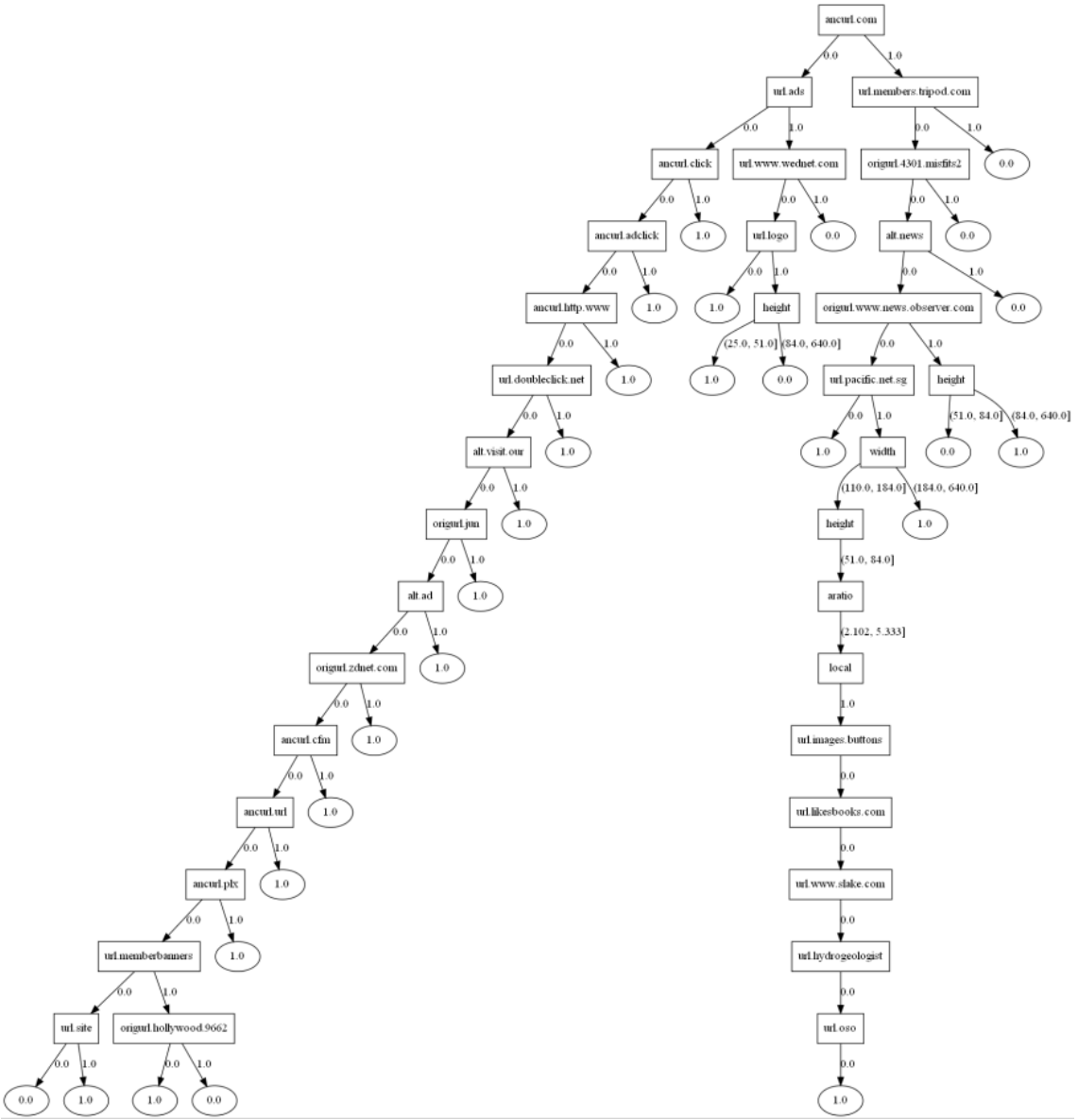


Media de las evaluaciones de la clase 1.0 (Gráfica)

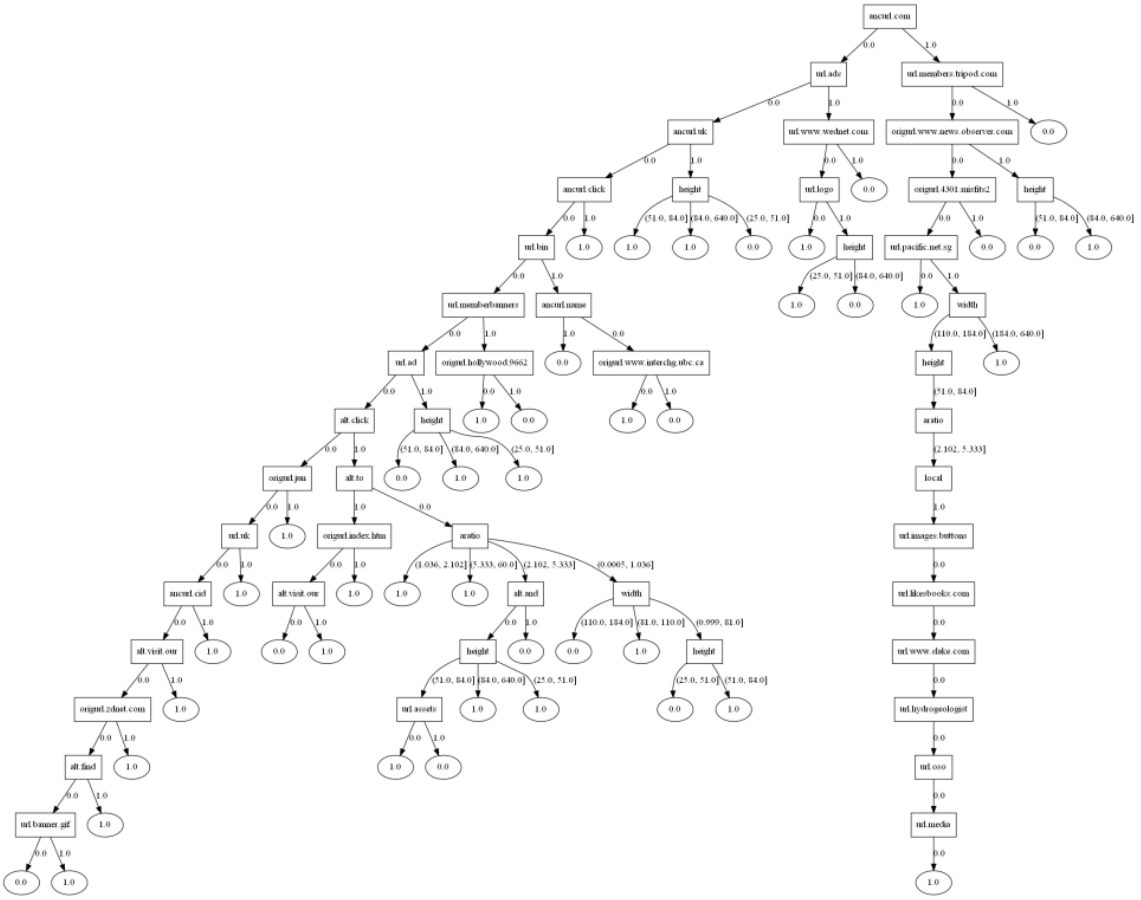
Vamos a ver ahora ejemplos de los árboles de decisión generados por nuestro programa con los distintos algoritmos de separación:



Árbol creado con algoritmo ID3



Árbol creado con C4.5



Árbol creado con Gini

Problemas encontrados

El principal problema encontrado es que el *dataset* contiene 1558 columnas. Las instancias no son tan problemáticas porque nos dan información extra y nos ayudan a entrenar mejor nuestro modelo, pero un número muy alto de columnas requiere comprobar si estas son útiles o no y posteriormente ejecutar todos los algoritmos de separación por todas ellas. Se hubiese podido abarcar más práctica si el *dataset* hubiera estado mejor en relación con el número de columnas, ya que se hubiese gastado mucho menos tiempo en la ejecución del programa.

Otro problema que hemos tenido ajeno a la ejecución ha sido la baja de la asignatura de uno de nuestros compañeros, teniendo los otros dos una carga de trabajo mayor de la que ya teníamos por las otras asignaturas.

Problemas en la implementación no ha habido en exceso, todo se ha ido implementando a un ritmo constante sin parones en una sección en concreto. Pero una cosa a puntualizar es que muchas de las funciones a implementar eran funciones recursivas. Este tipo de función ha sido siempre un reto para nosotros ya que estamos acostumbrados a programar iterativamente y una función recursiva siempre supone un reto para la planificación y la implementación.

Conclusiones

El programa implementado ha resultado ser muy robusto, con unos módulos independientes que nos aseguran una correcta ejecución sea cual sea el método que escojamos. Otro punto fuerte del programa implementado es que es un modulo "universal", de manera que puede ser implementado con toda clase de *dataset*, amoldándose a sus características ya que considera el mejor método de los que se ejecutan según las puntuaciones tales como el *recall,* la *precision* o el *f1-score.*

Gracias a estos algoritmos de separación implementados "a mano" que podrían ser implementados fácilmente importando módulos como si del *sklearn* se tratase, ganamos un entendimiento superior de lo que de verdad está ocurriendo en el programa.

Además, al generar una visualización correcta del *Decision Tree* final, vemos como el programa genera este árbol de manera gráfica y nos hace entender mejor el modelo creado.

Como aspectos negativos, deberíamos estudiar si haciendo un mejor uso de las librerías *numpy* para hacer cálculos sobres las matrices ayudaría a mejorar el cómputo sobre *datasets* con un numero muy alto de columnas, como es nuestro caso.

En relación con lo anterior y como hemos comentado en la sección de problemas, el tiempo que ocupa la generación de modelos es demasiado grande y no ha favorecido a la hora de analizar e implementar el proyecto, ya que el tiempo de ejecución era cercano a las 5 horas. Esto podría mejorar con un *dataset* con menos columnas.

En relación con los resultados ID3 nos arroja unos valores de precisión iguales a los de Gini y únicamente un poco por debajo de C4.5. Además, nos otorga los mejores valores para *recall*. Siendo además muy equilibrado obtiene como es de esperar, el mejor valor para la métrica f1-score tanto en la clase 0 como en la clase 1 (a la que le damos la mayor importancia). Es por esto que si tuviéramos que usar alguno de nuestros modelos entrenados para clasificar muestras de este *dataset*, nos inclinaríamos por usar ID3 como criterio de partición en nuestros árboles.