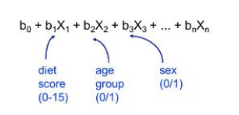
**Evaluación de Modelos y Feature Importance**

Además de ocuparnos de optimizar los modelos, es importante entender qué features del dataset contribuyen a este resultado y cuál es la relevancia de cada uno.

IE: Dado un dataset con 1000 features: Cuál(es) sirve(n)? Cómo identificarla(s)? Cómo medir su importancia?

**Modelos paramétricos**: A partir de una clasificación conocida, el problema se reduce a estimar los parámetros que mejor ajusten al dataset. IE: En regresión logística, **cada parámetro** está **asociado a** una **feature determinada**. En caso de tener **features estandarizadas**, el **tamaño de cada parámetro** nos puede dar una **idea** de la **importancia relativa de cada feature**.



Si trabajamos con árboles de clasificación, no tenemos parámetros para ajustar. Entonces hay que tomar otro camino para encontrar la importancia de las features en estos casos.

**Importancia de las Features en Árboles CART**:

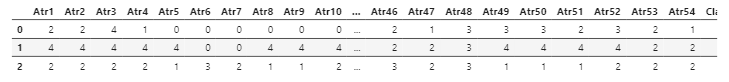
**En Python:** Vamos a trabajar con un dataset para la predicción de divorcios: Se hicieron 54 preguntas a personas. La Clase es si la persona está en pareja (0) o divorciada (1). Atr1 a Atr 54 son afirmaciones que se responden entre 0 y 4; siendo 0 que está totalmente de acuerdo y 4 que no coincide en absoluto. 1 a 3 son valores intermedios.

df = pd.read\_csv(‘../Data/divorce.csv’)

print(‘Filas:’, df.shape[0], ‘Columnas:’, df.shape[1])



df.head()



X = df.drop([‘Class’], axis=1)

y = df[‘Class’]

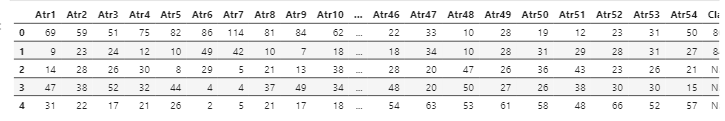
y.value\_counts(normalize=True)



Las clases están balanceadas.

Podemos ver la cantidad de respuestas por afirmación y por valor:

df.apply(pd.value\_counts)



# No dividimos en train y test porque no nos interesa evaluar la performance en este caso:

from sklearn import tree

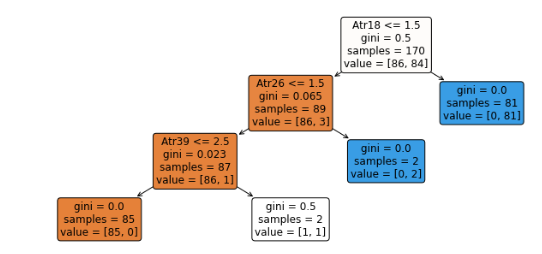
dt = tree.DecisionTreeClassifier(criterion = ‘gini’, max\_depth = 4, min\_samples\_leaf = 2)

dt.fit(X,y)

plt.figure(figsize=(12,5))

tree.plot\_tree(dt, feature\_names = df.columns[:,-1], filled=True, rounded=True, fontsize = 12)

plt.show()



Atr 18: “Mi conyuge y yo tenemos ideas similares sobre cómo debería ser el matrimonio”: Si se responde entre 2 y 4, automáticamente lo clasifica como divorciado.

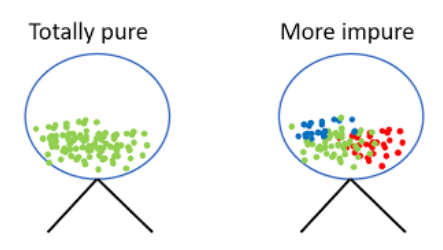
En caso de coincidir, pasa a ver Atr 26: “Conozco las ansiedades básicas de mi conyuge”. Si se responde entre 2 y 4, automáticamente lo clasifica como divorciado.

En caso de coincidir, pasa a ver Atr. 39: “Cuando lo necesitamos, podemos tomar nuestras discusiones con mi conyuge desde el principio y corregirlo”. Si responde entre 0 y 3, lo clasifica como casado. Si responde con 4, lo clasifica como divorciado.

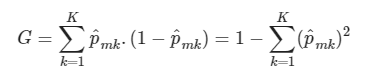
En cada nodo vemos:

* El criterio de partición
* El criterio de impureza (Gini = 0.5)
* El total de casos asignados al nodo.
* Los casos por clase.
* La clase seleccionada.

Para particionar cada nodo, se busca que los nodos hijos sean lo más puros posibles: Un nodo puro es aquél que contiene predominantemente observaciones de una misma clase.



Para medir la pureza, usamos el **índice de impureza de Gini**:



Siendo pmk la proporción de observaciones de entrenamiento de la clase k en la región m. Si es cercano a 0, entonces el nodo contiene mayoritariamente observaciones de una misma clase. Si es 0.5, entonces tiene máxima impureza. IE: El primer Nodo del algoritmo tiene un Gini de 0.5 y el subsiguiente tiene un Gini de 0.065. De hecho, la gran mayoría de las observaciones de este nodo es de la clase 0 (86 de 89 casos).

**En Python:**

total\_observaciones\_clase = pd.Series(y).value\_counts()

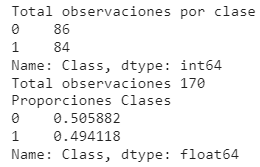
print(‘Total observaciones por clase”); print(total\_observaciones\_clase)

total\_observaciones = sum((total\_observaciones\_clase))

print(‘Total observaciones’, total\_observaciones)

proporciones\_clases = total\_observaciones\_clase / total\_observaciones

print(‘Proporciones clases’); print(proporciones\_clases)



G = 1 – sum(proporciones\_clases \*\*2)

print(“G: “, round(G,2))



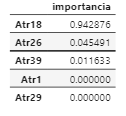
# Para calcular la importancia de las features se puede usar la propiedad **feature\_importances\_** de **DecisionTreeClassifier**.

Como se usaron sólo 3 features para generar las particiones, sólo estas 3 features tienen alguna importancia.

importancia\_features = pd.DataFrame(dt.feature\_importances\_, index = X.columns, columns = [‘importancia’])

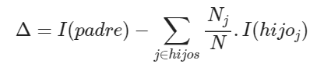
importancia\_features\_sort = importancia\_features.sort\_values(‘importancia’, ascending = False)

importancia\_features\_sort[0:5]



Lo que hace la **importancia de las features** es **comparar** la **impureza antes y después** de la partición. **Importancia de Gini o Ganancia**. Cuanto **mayor** sea la **ganancia**, **mayor será la importancia** de la feature: Si la impureza del nodo padre es mayor que la suma de impurezas de los hijos, entonces estamos mejorando la pureza.

**Reducción Total Normalizada del criterio Gini por feature:**



I: medida de impureza.

Nj: Cantidad de registros en el nodo hijo j.

N: Cantidad de registros en el nodo padre.

**En Python:**

ganancia\_gini\_Atr18 = 1000 \* 0.5 – (89/170) \* 0.065 – (81/170) \* 0.0000

ganancia\_gini\_Atr26 = (89/170) \* 0.065 – (87/170) \* 0.023 – (2/170) \* 0.0000

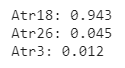
ganancia\_gini\_Atr39 = (87/170) \* 0.023 – (85/170) \* 0.0000 – (2/170) \* 0.5

norm = ganancia\_gini\_Atr + ganancia\_gini\_Atr26 + ganancia\_gini\_Atr39

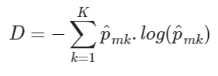
print(“Atr18”, round(ganancia\_gini\_Atr18 / norm,3))

print(“Atr26”, round(ganancia\_gini\_Atr26 / norm,3))

print(“Atr3”, round(ganancia\_gini\_Atr3 / norm,3))



También es posible particionar usando el criterio “entropy”. La **entropía** es, en **teoría de la información**, una **forma de medir** el **grado de desorganización** de un **sistema**. Lo que mide es qué tan parecidos son los elementos de un sistema, siendo 0 cuando son totalmente iguales y 1 cuando tiene el mayor grado de desorden posible.



Siendo pmk la proporción de observaciones de entrenamiento de la clase k en la región m.

Ahora vamos a generar un árbol para el mismo dataset, pero cambiando el criterio por entropía. **No necesariamente ambos criterios (Gini o entropía)** generan el mismo modelo:

from sklearn import tree

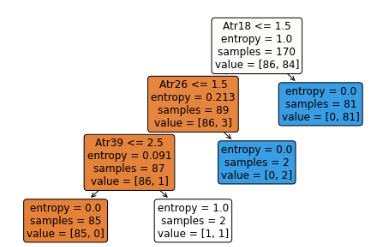
dt\_entropy = tree.DecisionTreeClassifier() criterion = ‘entropy’, max\_depth = 4, min\_samples\_leaf = 2)

dt\_entropy.fit(X, y)

plt.figure(figsize=(8,5))

tree.plot.tree(dt\_tentropy, feature\_names = df.columns[:,-1], filled = True, fontsize = 12)

plt.show()



Si conocemos las features más importantes, podemos generar un árbol más eficiente en términos de cómputo. Podemos reducir el conjunto de features en uno más chico con bajo o sin costo en la performance.

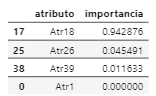
**En Python:**

dt = tree.DecisionClassifier(criterion = ‘gini’, max\_depth = 4, min\_samples\_leaf = 2)

dt.fit(X,y)

if\_select = pd.DataFrame({‘atributo’’:X.columns, ‘importancia’: dt.feature\_importances\_})

if\_select.sort\_values(‘importancia’, ascending = False).iloc[0:4, :]



features\_select = if\_select.atributo.values[if\_select.importancia.values>0]

features\_select



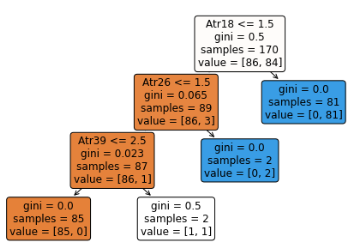
dt\_select = tree.DecisionTreeClassifier(criterion = ‘gini’, max\_depth = 4, min\_samples\_leaf = 2)

dt\_select.fit(X.loc[:, features\_select], y)

plt.figure(figsize=(8,5))

tree.plot\_tree(dt\_select, features\_names = features\_select, filled = True, rounded = True, fontsize = 12)

plt.show()



**Importancia de las Features en Ensambles:**

Vamos a generar un modelo de Random Forest sobre el mismo dataset de Divorcios y calcular la importancia de las features. La misma se calcula como la **media de la** **importancia de las features de los árboles base**. Random Forest aplica la técnica de bagging pero en vez de usar todas las variables independientes para cada modelo, lo que hace es aplicar muestreos con reposición. En este ejemplo, genera 10 datasets (n\_estimators = 10) donde cada uno es un muestreo de la features tomadas de a M sobre el total P de features (54). Entonces, la cantidad de features con importancia mayor a 0 será mayor que la del modelo CART, ya que se generan más árboles que tienen distintas features.

**En Python:**

from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(class\_weight = ‘balanced’, random\_state = 17, n\_estimators = 10)

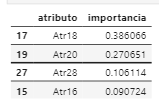
rf.fit(X, y)



Ahora vamos a calculary la importancia de las features, con la propiedad **feature\_importances\_**

rf\_select = pd.DataFrame({‘atributo’:X.columns, ‘importancia’: rf.feature\_importances\_})

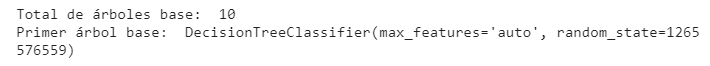
rf\_select.sort\_values(‘importancia’, ascending = False).iloc[0:4, :]



Ahora vamos a comprobar que la importancia de las features en los ensambles es el promedio de la importancia de los árboles base. Para esto vamos a usar la propiedad **estimators\_** para obtener los árboles base del ensamble.

print(‘total de árboles base: ‘, len(rf.estimators\_))

print(‘Primer ábrol base:’, rf.estimators\_[0])



Cada árbol base tiene su matriz de importancia de features. Esta matriz cuenta con 54 elementos, cada uno de estos elementos representa la importancia de una feature.

len(rf.estimators\_[0].features\_importances\_)



Veamos la importancia de la feature Atr18 a partir de los árboles base. Los atributos se llaman Atrx con x entre 1 y 54. El elemento 17 en la matrices de importancia de cada árbol base corresponde al valor para Atr18.

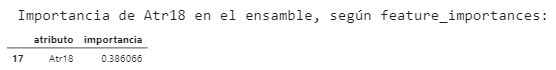
suma = sum([rf.estimators\_[i].feature\_importances\_[17] for i in range(len(rf.estimators\_))])

print(‘Importancia calculada de Atr18 en el ensamble: ‘, round(suma/10,6))



print(‘Importancia de Atr18 en el ensamble, según feature importances: ’)

rf\_select[re\_select.atributo == ‘Atr18’]



**Interpretabilidad de los Modelos:**

Además del resultado, nos interesa conocer cómo interpretarlos. A veces es posible y a veces no.

**LIME** (Local Interpretable Model-agnostic Explanations) es una biblioteca de Python que explica cómo decide un modelo de una forma comprensible, generando explicaciones a nivel local.

Vamos a seguir usando el dataset de divorcios, limitándolo a las 7 primeras features con mayor importancia en el modelo Random Forest:

**En Python:**

if\_select = pd.DataFrame({‘atributo’:X.columns, ‘importancia’: rf.feature\_importances\_})

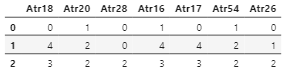
features = if\_select.sort\_values(‘importancia’, ascending = False).iloc[0:7,:][‘atributo’].values.tolist()

features



X\_lime = X[features]

X\_lime.head()



Se separan los datos en train y test. Se crea un modelo de ensamble Random Forest, porque necesitamos conocer las predicciones.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_lime, y, test\_size = 0.3, random\_\_state = 123)

rc\_exp = RandomForestClassifier(n\_estimators = 50, random\_state = 123)

rc\_exp.fit(X\_train, y\_train)



Ahora vamos a instanciar LimeTabularExplainer para poder generar explicaciones sobre una predicción determinada:

X\_train\_explainer = np.array(X\_train)

explainer = LimeTabularExplainer(X\_train\_explainer, mode = ‘classification’, training\_labels = y\_train, feature\_names = X\_train.columns, categorical\_features = list(range(7)), discretize\_continuous = False)

LimeTabularExplainer usa los siguientes parámetros:

**training\_data**: Es X\_train convertido en un numpy 2d array.

**mode** “classification” o “regression”, en este caso vamos a usar classification.

**training\_labels** variable target en train

**feature\_names** lista de índices de las columnas categóricas

**discretize\_continuous** Si es True, se discretizan todas las features no categóricas

**discretizer:** Tipo de discretización: quartile, decile, o entropy.

El objeto **explainer** y el método **explain\_instance** pueden usarse para generar explicaciones sobre una predicción concreta.

IE: Sobre la fila de índice 47, explain\_instance toma como argumentos: **data\_row** (fila en análisis, como en numpy array de 1d; **predict\_fn**, que es un clasificador que devuelve los valores de la predicción como probabilidades: **classifier.predict\_proba()** o **regressor.predict()**, según si se trata de una clasificación o de una regresión, respectivamente. **num\_features**, todas las features presentes en la predicción. El **output** es una instancia de tipo **explanation**.

**En Python:**

i = 47

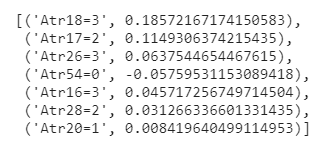
data\_row = np.array(X\_test.iloc[i])

explanation = explainer.explain\_instance(data\_row, rc\_exp.predict\_proba, num\_features = len(data\_row))

La clase **explanation** provee métodos para analizar y visualizar el resultado. Con **as\_list** se puede conseguir una lista (feature, peso) correspondiente a la predicción de la fila:

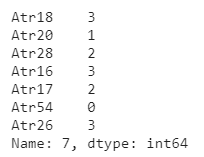
**En Python:**

explanation.as\_list()



En la columna de las features se pueden ver los valores que tienen dichas features en la fila 47:

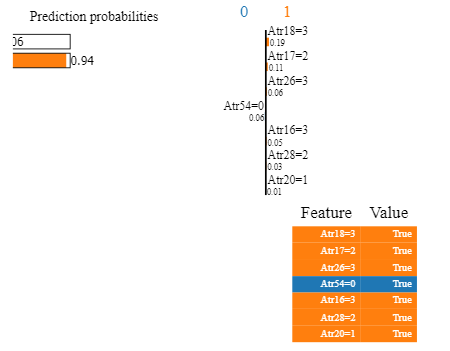
X\_test.iloc[47]



Con **show\_in\_notebook** podemos ver un gráfico que resume la información:

**En Python:**

explanation.show\_in\_notebook(show\_table = True)



Se trata de un gráfico de 3 partes:

A la izquierda se ve la **probabilidad predicha por el modelo** para el registro de índice 47.

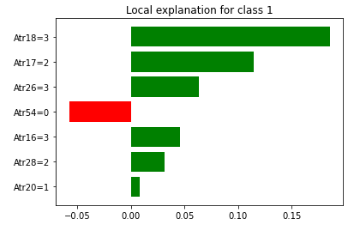
El panel del medio muestra las **features por orden de importancia**. Las naranjas son compatibles con la clase 1, las azules con la clase 0.

El panel de la derecha muestra de nuevo las features y sus valores en el registro de índice 47.

**as\_pyplot\_figure** muestra un gráfico de barras similar al panel central de show\_in\_notebook:

**En Python:**

explanation.as\_pyplot\_figure()



**Conclusiones:**

* Con el **índice de impureza Gini** y la **importancia de Gini** es posible realizar el Split de un nodo en nodos más puros.
* En el algoritmo **CART** se aplican los criterios **Gini** o **Entropía**.
* La **importancia** de las features es la **reducción total normalizada del criterio (Gini) por feature**. Cuánto gana en pureza al pasar del nodo padre a los nodos hijos.
* La **importancia en modelos de ensamble**, se calcula como la **media de la importancia de las features de los árboles base.**
* **Si sabemos** cuáles son las **features más importantes**, **podemos reducir el conjunto** de features del modelo y así **disminuir** su **costo computacional**.
* **Lime** es una **Librería de Python** que explica cómo se generó la predicción de una observación determinada.