PRÁCTICA 8
ALMACÉNS E MINARÍA DE DATOS

Álex Baquero DomÍnguez
alex.baquero@rai.usc.es

Hugo Gómez Sabucedo
hugo.gomez.sabucedo@rai.usc.es

Alejandro Aybar Cifuentes
alejandro.aybar@rai.usc.es

Índice

1. Clasificación.

Para este primer apartado, se nos proporcionan dos conjuntos de datos, sobre los cuales aplicaremos algunos algoritmos de clasificación, para comparar el resultado obtenido por los mismos. En este caso, hemos elegido aplicar tres algoritmos distintos, para ver cuál de ellos ofrece el mejor resultado. Analizaremos el algoritmo de los árboles de decisiones o Decision Tree, el de las máquinas de vectores de soporte (SVM) y el Random Forest.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

SEMILLA = 123456789
np.random.seed(SEMILLA)

dtrain = pd.read_csv('./clasificar_train.csv')
dtest = pd.read_csv('./clasificar_test.csv')

X_train = dtrain.drop(columns=['X', 'Y', 'id', 'Cat'])
X_test = dtest.drop(columns=['X', 'Y', 'id', 'Cat'])
y_train = dtrain["Cat"]
y_test = dtest["Cat"]
```

En primer lugar, lo que haremos será importar los datos como dataframes de pandas, y seleccionar aquellas columnas que nos interesen. Tras realizar los imports necesarios, en primer lugar estableceremos una semilla, de forma que los resultados serán siempre los mismos en todas las ejecuciones. Importamos, por separado, en dos dataframes distintos, los datos de train y de clasificación. A continuación, vamos a separar estos dataframes en los atributos independientes (de entrada) y dependientes (de salida). En este caso, los atributos dependientes (y) será la columna “Cat”, que es la que contiene la categoría de lo que sería la parcela (si es mar, playa, carretera, bosque…). Por otra parte, para obtener las variables independientes (que son los valores de las bandas que se nos indican en el guión; es decir, “landsat__1”, “landsat__2”, etc. hasta 7), lo que hacemos es quitar del dataframe las columnas que no nos interesan (la X y la Y, que son coordenadas geográficas; el id; y la categoría).

A continuación, vamos a aplicar los distintos algoritmos que hemos seleccionado.

1.1. Decision Tree

El algoritmo de árbol de decisión se basa en tres elementos principales: los nodos, que evalúan un atributo; las ramas, que representan la salida del nodo y la regla de decisión; y las hojas, que son las que representan la etiqueta de la clase (es decir, la salida). Su funcionamiento es el siguiente:

1. En primer lugar, seleccionar el mejor atributo para dividir los datos, empleando métricas de selección de atributos como pueden ser la ganancia de información o el ratio de Gini.
2. Seleccionar ese atributo como un nodo de decisión y dividir el dataset en subsets más pequeños.
3. Construir el árbol recursivamente, repitiendo estos pasos hasta que se cumpla una de las siguientes condiciones:
    a. Todas las tuplas pertenecen al mismo atributo.
    b. No hay atributos restantes.
    c. No hay más instancias.

```
18  #----------Decision Tree----------
19  from sklearn.tree import DecisionTreeClassifier
20  from sklearn import tree
21
22  #Creamos el clasificador
23  classifierDT = DecisionTreeClassifier(random_state=SEMILLA)
24
25  #Entrenamos el clasificador
26  classifierDT.fit(X_train, y_train)
27
28  #Realizamos la prediccion
29  predictionsDT = classifierDT.predict(X_test)
30
31  #Mostramos el árbol de decisión
32  fig = plt.figure(figsize=(40, 40))
33  _ = tree.plot_tree(classifierDT, filled=True)
```

Con el anterior código, aplicamos el algoritmo del árbol de decisión. Para ello, debemos importarlo desde el paquete sklearn.tree, y crear nuestro clasificador de tipo DecisionTreeClassifier, al cual aplicamos como random_state la semilla definida anteriormente, para poder replicar los mismos resultados siempre. A continuación, con el método .fit(), entrenamos el clasificador, empleando para ello la variable X e Y de train. Una vez entrenado, podemos realizar la predicción con .predict(), empleando los datos de test de las variables independientes, y guardando los resultados en una variable. Por último, mostramos el árbol de decisión, obteniendo la siguiente imagen.

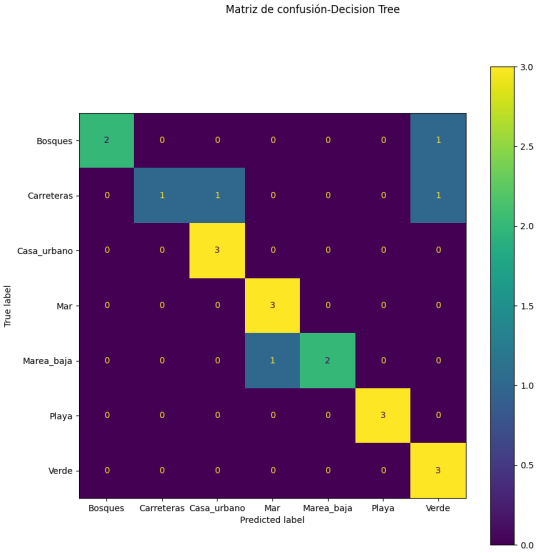
Para analizar la eficacia de este algoritmo, empleamos el siguiente código. Este será el mismo en el caso de los tres algoritmos, cambiando únicamente los nombres de las variables, por lo que no lo incluiremos en los siguientes casos.

```
#Analizamos la eficacia del método
from sklearn.metrics import accuracy_score, classification_report, plot_confusion_matrix
print("Decission Tree:")
print("  Accuracy: " + str(accuracy_score(y_test, predictionsDT)))
print(classification_report(y_test, predictionsDT))

fig, ax = plt.subplots(figsize=(10, 10))
disp = plot_confusion_matrix(classifierDT, X_test, y_test, ax=ax)
disp.figure_.suptitle("Matriz de confusión-Decision Tree")
plt.show()
```

Con esto, obtenemos el siguiente resultado. Vemos que el algoritmo tiene una precisión global del 80.95%, lo cual es un resultado bastante aceptable. Esto significa que, globalmente, casi el 81% de los datos se predijeron correctamente. Con la función classification_report podemos ver estos datos detallados por clases. Podemos ver que, en clases como los bosques o las carreteras, todos los datos son predichos correctamente, mientras que en el caso de Verde, solo el 60% se predicen de forma correcta. Sin embargo, debemos tener en cuenta que nuestro dataset de entrenamiento es relativamente pequeño, con solo 46 ejemplos. De tener más ejemplos, quizás el algoritmo se entrenaría mejor.

Por otra parte, tenemos el recall y el f1-score. El recall indica cuántos positivos se predijeron correctamente con el modelo. Se calcula dividiendo los verdaderos positivos (los datos que el clasificador identificó correctamente) por el total de miembros de dicha clase. El f1 combina la precisión y el recall, de forma que si ambos son altos, f1 será alto; si son bajos, será bajo; y si uno es alto y otro bajo, f1 será bajo. Es una buena forma de distinguir si el clasificador es realmente bueno, o si está encontrando atajos a la hora de identificar los miembros de las clases.

```
Decission Tree:
  Accuracy: 0.8095238095238095
              precision    recall  f1-score   support

     Bosques       1.00      0.67      0.80         3
  Carreteras       1.00      0.33      0.50         3
 Casa_urbano       0.75      1.00      0.86         3
         Mar       0.75      1.00      0.86         3
  Marea_baja       1.00      0.67      0.80         3
      Playa        1.00      1.00      1.00         3
      Verde        0.60      1.00      0.75         3

    accuracy                           0.81        21
   macro avg       0.87      0.81      0.79        21
weighted avg       0.87      0.81      0.79        21
```



Por último, tenemos la matriz de confusión. Esta es una matriz nxn, con n=número de clases, donde se representa, en las filas, la etiqueta real de la clase y, en las columnas, la predicha por el modelo. De esta forma, podemos ver que, por ejemplo, para la clase Mar, los 3 datos se predijeron como Mar. Sin embargo, en la clase Bosques, 2 datos se predijeron correctamente, mientras que uno se predijo como Verde. En la clase Carreteras, únicamente 1 dato se predijo correctamente; los otros dos se predijeron, uno como Carreteras, y otro como Verde.

En general, Decision Tree es un algoritmo que funciona correctamente, aunque podemos ver que en clases que pueden ser parecidas (por ejemplo, Verde Bosques, o Marea Baja y Mar) confunde algunos de los elementos.

1.2. SVM

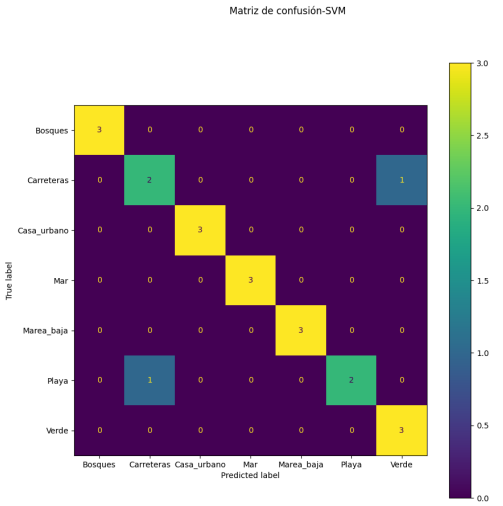
El algoritmo SVM (Support Vector Machine) se basa en construir un hiperplano en un espacio multidimensional, para separar las diferentes clases. Este algoritmo genera, de forma iterativa, el hiperplano óptimo, que se usa para minimizar un error. Su idea básica es encontrar el Maximal Marginal Hyperplane, que es el que divide mejor el dataset en las diferentes clases. Para encontrarlo, sigue los siguientes pasos:

1. Genera hiperplanos que separan las clases de la mejor forma posible.
2. Elegir el hiperplano adecuado, con la máxima segregación de los datos más cercanos al mismo.

```python
#----------Support Vector Machine classifier----------
from sklearn.svm import SVC

#Creamos el clasificador
classifierSVM = SVC(kernel="linear", random_state=SEMILLA)

#Entrenamos el clasificador
classifierSVM.fit(X_train, y_train)

#Realizamos la prediccion
predictionsSVM = classifierSVM.predict(X_test)
```

El algoritmo se aplica de la siguiente forma. Tras importar el módulo, creamos el clasificador empleando el constructor SVC, con un kernel lineal (es decir, que usa el producto vectorial para transformar los puntos) y con el random_state definido antes. A continuación, de forma similar a como hicimos con el Decision Tree, hacemos el .fit() para entrenarlo, y hacemos las predicciones con el .predict().

```
SVM:
  Accuracy: 0.9047619047619048
              precision    recall  f1-score   support

     Bosques       1.00      1.00      1.00         3
  Carreteras       0.67      0.67      0.67         3
 Casa_urbano       1.00      1.00      1.00         3
         Mar       1.00      1.00      1.00         3
  Marea_baja       1.00      1.00      1.00         3
      Playa        1.00      0.67      0.80         3
      Verde        0.75      1.00      0.86         3

    accuracy                           0.90        21
   macro avg       0.92      0.90      0.90        21
weighted avg       0.92      0.90      0.90        21
```



De esta forma, si visualizamos los resultados obtenidos, podemos ver que este algoritmo tiene una precisión del 90.48%, lo cual es casi 10 puntos superior a la que tenía el anterior. Además, únicamente falla en dos clases: Carreteras, en donde tiene una precisión de 0.67, y Verde, con una precisión de 0.75. El recall es, en ambos casos, de 0.67. Esto significa que se predijeron correctamente el mismo número de valores en ambos casos. Si nos fijamos en la matriz de confusión, vemos que esto es así. Tanto para las Carreteras como para Verde se predijeron 2 datos correctamente, mientras que el otro se predijo, en el primer caso, como Verde; y, en el segundo, como Carretera. Es decir, el modelo ha confundido un dato de una de estas clases con el de la otra, y viceversa.

En general, aunque la precisión sea del 90%, podemos deducir que es un buen algoritmo, y bastante mejor que el anterior. Es importante tener en cuenta, como ya dijimos, que al tener una pequeña cantidad de datos de entrenamiento, y también de test, un único error va a hacer que disminuya en gran medida la precisión del

algoritmo (en este caso, con sólo 3 datos por clase, un error en la predicción de un dato en una clase hace, como vemos, que la precisión de dicha clase sea del 67%.

1.3. Random Forest

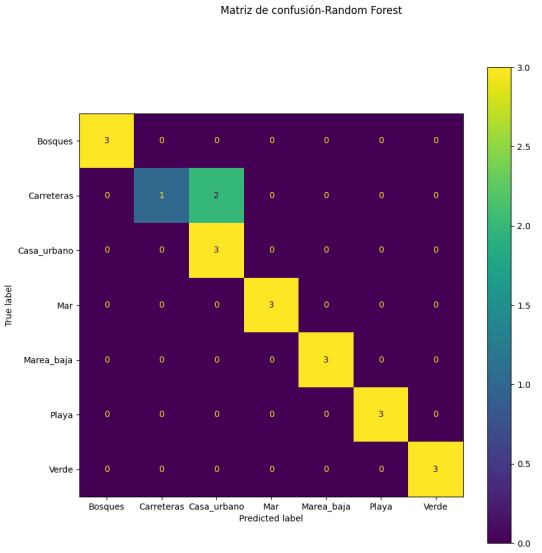
El último algoritmo es el de Random Forest. Este algoritmo se basa en el de los árboles de decisión: construye múltiples árboles, basándose en diferentes ejemplos de los datos, y obtiene la predicción de un nuevo dato mediante una agregación de las predicciones de todos los árboles individuales. El proceso que sigue este algoritmo es el siguiente:

1. Selecciona datos aleatorios del dataset.
2. Construye un árbol de decisión para cada ejemplo, y obtiene una predicción del resultado de cada uno de ellos.
3. Da un voto, o un peso, a cada uno de los resultados predichos.
4. Selecciona el resultado de la predicción con más votos como resultado final.

```
72  #----------Random Forest Classifier----------
73  from sklearn.ensemble import RandomForestClassifier
74  #Creamos el clasificador
75  classifierRF = RandomForestClassifier(n_estimators=3, random_state=SEMILLA)
76
77  #Entrenamos el clasificador
78  classifierRF.fit(X_train, y_train)
79
80  #Realizamos la prediccion
81  predictionsRF = classifierRF.predict(X_test)
```

En este caso, importamos desde sklearn.ensemble el RandomForestClassifier, y construimos el clasificador con este método, usando el random_state explicado anteriormente, y con un número de estimadores igual a 3. Este parámetro representa el número de árboles que vamos a construir antes de elegir el resultado final. Hemos decidido fijar este parámetro a 3, aunque en nuestro caso, si quisiéramos ver cuál es el parámetro óptimo, podríamos probar uno a uno, hasta llegar al número máximo de datos del dataset (22). A continuación, entrenamos el clasificador y realizamos la predicción.

```
Random Forest:
  Accuracy: 0.9047619047619048
              precision    recall  f1-score   support

     Bosques       1.00      1.00      1.00         3
  Carreteras       1.00      0.33      0.50         3
 Casa_urbano       0.60      1.00      0.75         3
         Mar       1.00      1.00      1.00         3
  Marea_baja       1.00      1.00      1.00         3
      Playa        1.00      1.00      1.00         3
      Verde        1.00      1.00      1.00         3

    accuracy                           0.90        21
   macro avg       0.94      0.90      0.89        21
weighted avg       0.94      0.90      0.89        21
```



De esta forma, obtenemos el anterior resultado. La precisión es de 90.47%, idéntica al algoritmo de SVM, pero si nos fijamos en la precisión por clases, podemos ver que sólo ha fallado en una (Casa_urbano). Observando la matriz de confusión, podemos ver que uno de los elementos de esta clase se ha predicho como de la clase Carreteras, lo cual hace que disminuya la precisión en la clase Casa_urbano, por una parte, y el recall en la clase Carreteras, por otra.

Aun así, globalmente, es un buen algoritmo. Si, además, probamos con otros parámetros para el número de estimadores, vemos que hay casos, como con n=5 (el cual es un número de árboles generados relativamente pequeño), en donde la precisión es del 100%: es decir, que el clasificador funciona a la perfección, asignando todos los datos a la clase que corresponden.

En general, por todo lo que hemos observado, podemos observar que el mejor algoritmo es el de Random Forest. Comparar su rendimiento con el de Decision Tree no tiene mucho sentido, ya que en el fondo el Random Forest no es más que un ejecución de múltiples Decision Tree, obteniendo de ellos los que proporcionen la mejor solución. Si lo comparamos con el de SVM, vemos que ambos obtuvieron una precisión muy similar, pero el Random Forest tiene mejor precisión y recall en las clases individuales: únicamente se equivoca en una ocasión, y es debido al parámetro elegido de número máximo de árboles, ya que con otros parámetros no mucho más mayores, realiza predicciones perfectas.

Sin embargo, cabe notar que esto también puede deberse a que nuestro conjunto de datos es muy pequeño. Entrenamos al algoritmo con pocos datos, y también lo probamos con pocos datos. Esto puede dar lugar a que, si quisiéramos hacer muchos tests con el algoritmo poco entrenado, podríamos obtener muchos resultados erróneos. Pero, por el contrario, esto también puede jugar a nuestro favor en el caso de tener pocos tests, o tests que sean similares a los valores con los que se entrenó el algoritmo, ya que obtendríamos predicciones perfectas.

Globalmente, de entre los tres algoritmos elegidos, el mejor es, sin duda, Random Forest.

2. Regresión.

En primer lugar, vamos a descargar el CSV y a importarlo en Python. Para ello utilizaremos, como de costumbre siempre que trabajamos con dataframes, la librería pandas.

Por lo tanto, importamos el CSV y observamos las 5 primeras filas con *head(5)* y un resumen estadístico del dataframe con *describe()*.

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("./AMD_regresion.csv")

print(df.head(5))
print(df.describe())
```

Vamos a observar los resultados obtenidos:

```
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8> python .\regresion_random_forest.py
         LA        CL        SL        JD        ST        SS        CH        ML        U2        fC
0  0.197222  0.069186  0.753770  0.150234  0.125741  0.846285  0.001599  0.079746  0.686706  0.314781
1  0.197222  0.064823  0.746213  0.150234  0.131338  0.846475  0.001307  0.089313  0.685159  0.313859
2  0.197222  0.060592  0.738580  0.150234  0.144543  0.847232  0.001313  0.077258  0.650385  0.314817
3  0.197222  0.056495  0.730875  0.150234  0.152247  0.848040  0.001359  0.078295  0.650935  0.317044
4  0.197222  0.052533  0.723100  0.150234  0.151633  0.847885  0.001352  0.093972  0.673397  0.319883
                LA             CL             SL             JD             ST             SS             CH             ML             U2             fC
count  217649.000000  217649.000000  217649.000000  217649.000000  217649.000000  217649.000000  217649.000000  217649.000000  217649.000000  217649.000000
mean        0.563587       0.418609       0.409900       0.497218       0.549780       0.851131       0.005024       0.024025       0.252751       0.329922
std         0.216390       0.379739       0.301756       0.152238       0.268661       0.097127       0.014541       0.019427       0.128770       0.046559
min         0.063889       0.000019       0.000019       0.150234      -0.007706       0.002812       0.000000       0.002051       0.003107       0.005139
25%         0.413889       0.056495       0.108696       0.384886       0.332870       0.844810       0.000810       0.011605       0.162733       0.307067
50%         0.630556       0.246232       0.391779       0.516467       0.604639       0.865297       0.001624       0.018308       0.229494       0.330746
75%         0.719444       0.790352       0.691342       0.628311       0.793836       0.887000       0.003769       0.029788       0.319338       0.354296
max         0.952778       0.999981       0.999981       0.735768       0.962653       1.055845       0.739710       0.204929       0.996430       0.922035
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8>
```

Si nos fijamos, no parece que encontremos una gran cantidad de valores anómalos en el conjunto de datos. Solo encontramos valores 0 en la variable CH, y parece algo normal ya que se mueve en un rango cercano al 10e-2. Sin embargo, como el máximo (0.739) está bastante alejado del resto de valores, analizaremos los valores atípicos en esta variable.

Además, en SS, sí que encontramos que el máximo es 1.05. Esto no debería ser así, ya que los valores parecen estar todos normalizados en el intervalo [0,1]. Al mismo tiempo, vemos la presencia de un mínimo negativo en la variable ST, lo cual también incumple la normalización establecida.

Por lo tanto, vamos a proceder con la eliminación de valores atípicos en las variables ST, SS y CH.

Para la variable CH, vamos a aplicar la estrategia de excluir los valores atípicos extremos. Recordemos que los valores atípicos extremos son aquellos que distan 3 veces o más el rango intercuartílico por encima del tercer cuartil, o por debajo del primero.

Para las variables ST y SS, vamos a excluir del dataframe aquellos valores que se salgan del intervalo [0,1].

Veamos cómo implementamos esto en Python:

```python
# ------------------------------------------------------------------------
# ELIMINACION DE VALORES ATIPICOS

# obtener rango intercuartílico
Q3 = np.quantile(df.CH, 0.75)
Q1 = np.quantile(df.CH, 0.25)
IQR = Q3 - Q1

# umbrales extremos
umbral_bajo = Q1 - 3 * IQR
umbral_alto = Q3 + 3 * IQR

# variable CH -> eliminar valores atípicos extremos
CH_limpia = [x for x in df.CH if ((x > umbral_bajo) & (x < umbral_alto))]

# variables ST y SS -> eliminar valores fuera del rango [0,1]
ST_limpia = [x for x in df.ST if (x > 0.0)]
SS_limpia = [x for x in df.SS if (x < 1.0)]

# eliminamos los valores atipicos
df_limpio = df.loc[df.ST.isin(ST_limpia)]
df_limpio = df_limpio.loc[df.SS.isin(SS_limpia)]
df_limpio = df_limpio.loc[df.CH.isin(CH_limpia)]

print(df_limpio.describe())

```

Por lo tanto, veamos cómo queda el resumen estadístico del dataframe limpio una vez extraídos los valores atípicos:

```
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8> python .\regresion_random_forest.py
                 LA             CL             SL             JD             ST             SS             CH             ML             U2             fC
count  202296.000000  202296.000000  202296.000000  202296.000000  202296.000000  202296.000000  202296.000000  202296.000000  202296.000000  202296.000000
mean        0.554084       0.403001       0.413163       0.495871       0.562446       0.862755       0.002387       0.024677       0.256983       0.332007
std         0.213224       0.375558       0.303328       0.152459       0.268573       0.054785       0.002444       0.019735       0.130118       0.040417
min         0.063889       0.000019       0.000019       0.150234       0.000533       0.057708       0.000000       0.002542       0.003107       0.020576
25%         0.397222       0.056495       0.114188       0.382693       0.355513       0.846975       0.000764       0.011986       0.165641       0.309228
50%         0.619444       0.209649       0.391779       0.514274       0.627083       0.866840       0.001483       0.018937       0.232943       0.331508
75%         0.708333       0.761250       0.699376       0.628311       0.800933       0.888110       0.003086       0.030739       0.325349       0.354178
max         0.952778       0.999981       0.999981       0.735768       0.962653       0.996625       0.012646       0.204929       0.996430       0.720275
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8>
```

Como podemos ver, ya no encontramos valores fuera del intervalo [0,1] en ninguna variable. Además, los valores de la variable CH ya se encuentran en un rango aceptable.

El número de tuplas ha pasado a ser 202.296, en vez de 217.649. Es decir, hemos quitado unas 15.000 tuplas. Lo consideramos un balance positivo, ya que seguimos disponiendo de tuplas de sobra para realizar nuestro análisis, y nos hemos quitado los valores atípicos que existían.

Como todos los valores son de tipo numérico, no tenemos que aplicar ningún one-hot encoding sobre el conjunto de datos.

Con los datos ya preparados, podemos proceder a aplicar un Random Forest sobre la variable fC.

2.1. Aplicación de un Random Forest o una variante a la columna fC.

Separación en targets y features

En primer lugar, vamos a separar el conjunto de datos en targets y features. Targets son las variables a predecir, en este caso, fC. Features son aquellas variables de las que nos ayudamos para realizar la predicción.

Veamos cómo implementamos esto en Python:

```
35  # ------------------------------------------------------------------------
36  # RANDOM FOREST
37
38  # --------------------------------------------------------
39  # separamos features y targets
40
41  # sacamos los targets (fC) del conjunto de datos
42  labels = np.array(df_limpio.fC)
43  df_limpio = df_limpio.drop('fC', axis=1)
44
45  # guardamos la lista de features
46  lista_features = list(df_limpio.columns)
47  features = np.array(df_limpio)
48
49  # visualizamos los resultados
50  print(labels)
51  print(lista_features)
52  print(features)
53
```

Veamos si se han guardado los targets y features de manera correcta:

```
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8> python .\regresion_random_forest.py
[0.314781 0.313859 0.314817 ... 0.413034 0.432058 0.429065]
['LA', 'CL', 'SL', 'JD', 'ST', 'SS', 'CH', 'ML', 'U2']
[[0.197222 0.069186 0.75377  ... 0.001599 0.079746 0.686706]
 [0.197222 0.064823 0.746213 ... 0.001307 0.089313 0.685159]
 [0.197222 0.060592 0.73858  ... 0.001313 0.077258 0.650385]
 ...
 [0.802778 0.023142 0.349646 ... 0.       0.029622 0.507936]
 [0.808333 0.013815 0.383276 ... 0.       0.03917  0.485084]
 [0.813889 0.013815 0.383276 ... 0.       0.039005 0.446451]]
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8>
```

En rojo podemos ver los valores de target (fC), en verde la lista de features, y en amarillo los valores de dicha lista.

Separación en conjunto de entrenamiento y de testeo

A continuación, vamos a separar nuestro conjunto de datos, esta vez a nivel de filas, ya que distinguiremos un conjunto de entrenamiento y otro de testeo. Para ello, nos ayudaremos de la librería sk_learn, que nos ofrece la función train_test_split.

Para la separación, vamos a dedicar el 80% de las tuplas para el entrenamiento, y el 20% restante para el testeo.

```
50
51  # ------------------------------------------------------
52  # separamos conjunto de entrenamiento y de testeo
53
54  train_features, test_features, train_labels, test_labels = train_test_split(
55                                  features, labels, test_size=0.2, random_state=42)
56
57  print('Training Features Shape:', train_features.shape)
58  print('Training Labels Shape:', train_labels.shape)
59  print('Testing Features Shape:', test_features.shape)
60  print('Testing Labels Shape:', test_labels.shape)
```

Observemos el tamaño de cada uno de los conjuntos:

```
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8> python .\regresion_random_forest.py
Training Features Shape: (161836, 9)
Training Labels Shape: (161836,)
Testing Features Shape: (40460, 9)
Testing Labels Shape: (40460,)
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8>
```

Como podemos observar, de las 202.296 tuplas, utilizamos 161.836 para el entrenamiento y 40.460 para el testeo.

Entrenamiento y predicción

Con los datos ya separados adecuadamente, vamos a proceder con el entrenamiento del modelo y la posterior predicción.

En primer lugar, creamos el modelo de regresión Random Forest, con 10 árboles de decisión. Normalmente se utiliza un número más alto, pero por cuestiones de rendimiento hemos tenido que reducirlo.

Con el modelo creado, procedemos a entrenar el modelo, utilizando para ello el conjunto de entrenamiento que hemos designado. A continuación, predecimos el conjunto de testeo.

```
67  # ------------------------------------------------------
68  # entrenamiento y testeo
69
70  # iniciamos el regresor con 10 arboles de decision
71  rf = RandomForestRegressor(n_estimators = 10, random_state = 7)
72
73  # entrenamos el modelo con el conjunto de entrenamiento
74  rf.fit(train_features, train_labels)
75
76  # predecimos el conjunto de testeo
77  predictions = rf.predict(test_features)
78
```

Para saber cómo de exitosa ha sido la predicción, vamos a calcular el error absoluto medio y el porcentaje de precisión. Para el error absoluto medio, calculamos el valor absoluto de la diferencia entre las predicciones y los valores reales de fC. Para el porcentaje de precisión, dividimos este valor entre el valor real de fC y restamos el resultado a 100.

```
79  # calculamos los errores absolutos
80  errores = abs(predictions - test_labels)
81
82  # obtenemos el error absoluto medio
83  print('Error absoluto medio:', round(np.mean(errores), 2))
84
85  # calculamos el porcentaje de error absoluto medio
86  error_medio = 100 * (errores / test_labels)
87
88  # calculamos la precision de la prediccion
89  precision = 100 - np.mean(error_medio)
90  print('Precisión media:', round(precision, 2), '%.')
91
```

Observemos los resultados:

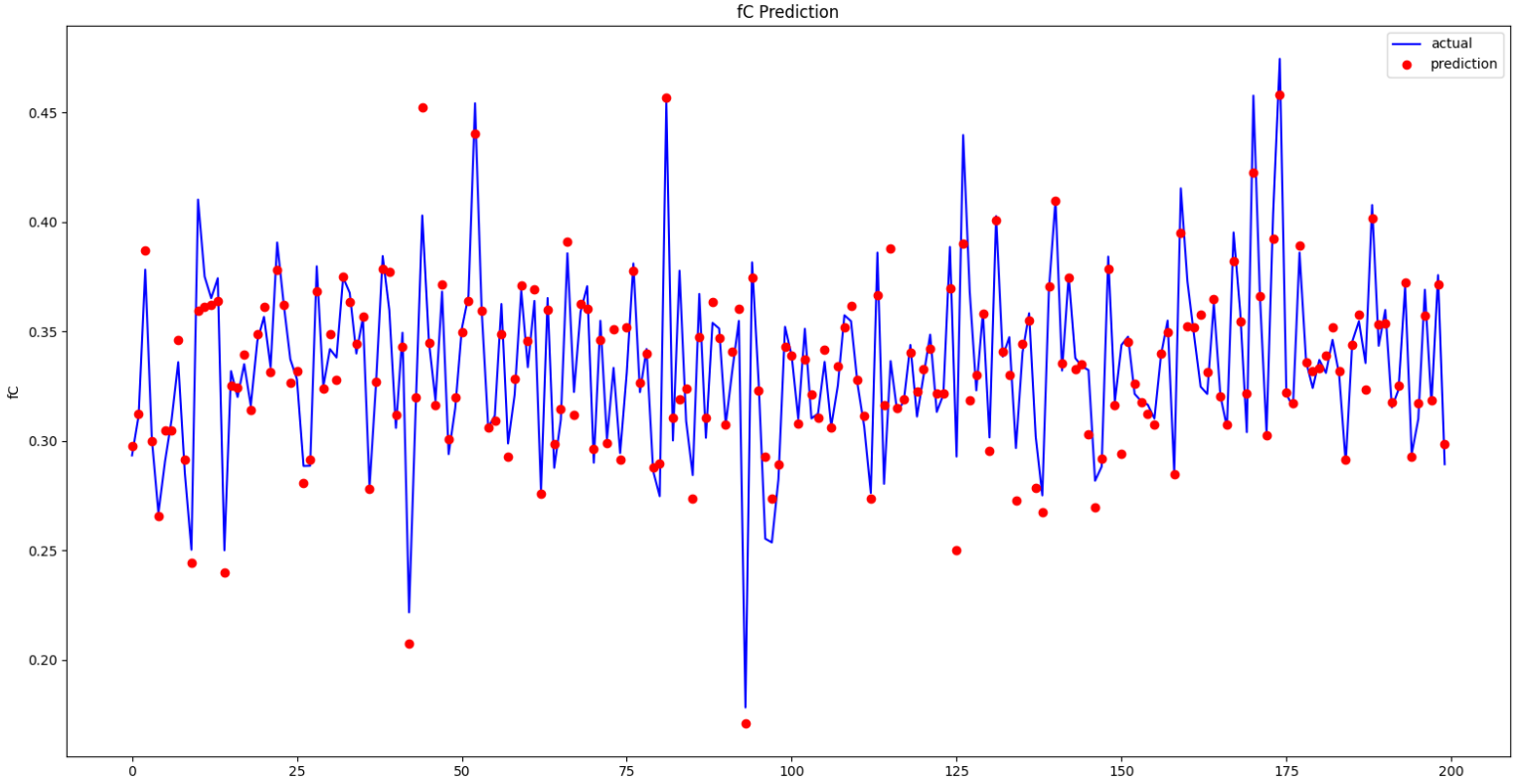
```
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8> python .\regresion_random_forest.py
Error absoluto medio: 0.01
Precisión media: 97.44 %.
```

Como podemos observar, el porcentaje de éxito es bastante alto. Un 97.44% de precisión indica que la predicción es muy fiable.

Veamos de forma visual cómo de buena ha sido la predicción. Como el conjunto de testeo tiene 40.460 valores, vamos a coger el subconjunto de los 200 primeros valores, para que podamos apreciar con detalle la diferencia entre predicción y realidad.

```
91  # ------------------------------------------------------
92  # visualizacion de resultados
93
94  # añadimos los valores reales y la prediccion al grafico
95  plt.plot(test_labels[:200], 'b-', label = 'actual')
96  plt.plot(predictions[:200], 'ro', label = 'prediction')
97
98  # mostramos la leyenda
99  plt.legend()
100
101 # ajustamos los titulos
102 plt.ylabel('fC')
103 plt.title('fC Prediction')
104
105 # mostramos el grafico
106 plt.show()
```

Veamos la salida del gráfico:



Como podemos observar, vemos que las predicciones tienden a ser bastante certeras, no habiendo una distancia muy grande entre el valor real y la predicción de éste.

Conclusión

Teniendo en cuenta el porcentaje de precisión (97.44%) y la visualización de los primeros 200 valores de la predicción y su distancia al valor real, podemos determinar que la regresión con Random Forest tiene una fiabilidad bastante alta.

El motivo de tal éxito lo podemos atribuir al alto número de valores con los que se contaba para el entrenamiento, siendo 161.836 las tuplas que han ayudado a entrenar el modelo. Por lo tanto, no ha sido un impedimento el determinar un número bajo de árboles de decisión (10 en este caso) de cara a obtener una predicción fiable.

2.2 Predicción sobre el entrenamiento de una red neuronal.

Metodología

En este caso, vamos a ayudarnos de una red neuronal para obtener la regresión con la que predecir el modelo.

Como ya hemos separado nuestros datos tanto en features y labels como en conjunto de entrenamiento y de testeo para la regresión con Random Forest, utilizaremos esta misma separación para este ejercicio.

Por lo tanto, nos adentramos directamente en el entrenamiento y la predicción de una regresión mediante una red neuronal. Para ello, nos ayudaremos de la red neuronal *keras*, que es una herramienta de la librería *tensorflow* muy utilizada en el mundo del deep learning.

Teniendo en cuenta que nuestra red neuronal tiene que adaptarse a un modelo de regresión, ajustaremos los parámetros de tal manera. Veamos cómo implementamos esto en Python:

Creación del modelo en Python

En primer lugar, creamos el modelo Sequential, que se encargará de agrupar una pila lineal de capas en un modelo keras, además de ofrecer el entrenamiento y la inferencia del modelo.

A continuación, añadimos 3 capas al modelo, que se introducirán en el tope de la pila de capas. Las 3 capas serán de tipo *Dense*, de manera que se encargarán de la activación elemento a elemento mediante el producto escalar del vector input y la matriz de pesos creada por la capa, sumándole un vector de bias, también creado por la capa.

La primera capa tendrá una dimensionalidad de 256 y marcará la forma del input (9,). Además, utilizará la función de activación de la unidad linear rectificada (ReLU), que aplicará el máximo entre 0 y el input tensor elemento a elemento.

Para la segunda capa, la dimensionalidad será de 128. La función de activación será la de por defecto del modelo Sequential.

Por último, tendremos una tercera capa donde la dimensionalidad será 1, ya que nuestro output se compone de 1 sola variable (fC).

```
65  # ------------------------------------------------------------------------
66  # keras
67
68  # creamos el modelo Sequential
69  model = Sequential()
70
71  # primera capa -> dimensionalidad 256
72  model.add(Dense(256, input_shape=(9,) , activation='relu'))
73
74  # segunda capa -> dimensionalidad 128
75  model.add(Dense(128))
76
77  # tercera capa -> dimensionalidad 1 (output)
78  model.add(Dense(1))
79
```

Con el modelo ya creado y con sus capas añadidas, vamos a compilarlo. Nos ayudaremos para ello de la función *compile,* a la que indicaremos que para la optimización se utilice el algoritmo *Adam*. Además, le indicamos que la función *loss* se realice mediante el método de *mean_squared_error*. Esto será importante para establecer nuestro modelo como uno de regresión.

```
79
80  # compilamos el modelo
81  model.compile(optimizer='Adam', loss='mean_squared_error')
82
```

Entrenamiento y testeo

Habiendo creado y compilado el modelo, podemos proceder con el entrenamiento y la predicción de éste. Para ello, utilizaremos las funciones *fit* y *predict*. Para el entrenamiento, utilizaremos un tamaño de bloque de 16, y haremos 4 barridas sobre el conjunto de datos. Además, utilizaremos los datos reservados para el testeo como datos de validación.

```
83  # entrenamos el modelo
84  model.fit(X_train, y_train, batch_size=16,epochs=4,verbose=1,validation_data=(X_test, y_test))
85  model.summary()
86
87  # predecimos el modelo
88  y_pred = model.predict(X_test)
89  print(y_pred)
90
```

Veamos cómo ha sido el rendimiento durante el entrenamiento del modelo y durante la predicción de éste.

Rendimiento del entrenamiento:

```
PS C:\Users\alex1\Documents\UNIVERSIDAD\AMD\practicas\p8> python .\regresion_red_neuronal.py
2022-12-17 19:30:15.446426: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary
nce-critical operations:  AVX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/4
10115/10115 [==============================] - 28s 3ms/step - loss: 0.0010 - val_loss: 8.2630e-04
Epoch 2/4
10115/10115 [==============================] - 26s 3ms/step - loss: 7.4475e-04 - val_loss: 7.3725e-04
Epoch 3/4
10115/10115 [==============================] - 24s 2ms/step - loss: 6.9034e-04 - val_loss: 6.1454e-04
Epoch 4/4
10115/10115 [==============================] - 26s 3ms/step - loss: 6.5197e-04 - val_loss: 6.0465e-04
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               2560

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 1)                 129

=================================================================
Total params: 35,585
Trainable params: 35,585
Non-trainable params: 0
```

Como podemos observar, se ha invertido alrededor de 25 segundos para cada barrida sobre el conjunto de datos. Además, si nos fijamos en el valor *loss*, vemos cómo a partir de la tercera barrida, el valor se mantiene estable. Esto quiere decir que podríamos haber omitido las dos últimas barridas sin haber afectado prácticamente a la calidad de nuestro modelo.

Veamos ahora el rendimiento durante la predicción:

```
_________________________________________________________________
1265/1265 [==============================] - 2s 2ms/step
[[0.3237797 ]
 [0.34391695]
 [0.36273766]
 ...
 [0.37356353]
 [0.33151537]
 [0.3052113 ]]
```

Para la predicción se han invertido apenas 2 segundos, lo cual es un tiempo bastante aceptable si la predicción ha sido fiable, cosa que mediremos a continuación.

Evaluación de los resultados

De manera similar a como hicimos con la predicción de Random Forest, vamos a obtener el porcentaje de precisión de la predicción calculando el error medio, que será el valor absoluto de la diferencia entre la predicción y el valor real. La precisión la obtendremos restando este valor a 100.

```
91   # calculamos los errores absolutos
92   errores = []
93
94   for i in range(0,len(y_pred)-1):
95     errores.append(abs(y_pred[i] - y_test[i]))
96
97   # obtenemos el error absoluto medio
98   print('Error absoluto medio:', round(np.mean(errores), 2))
99
100  # calculamos el porcentaje de error absoluto medio
101  error_medio = []
102
103  for i in range(0,len(errores)-1):
104    error_medio.append(abs(errores[i] - y_test[i]))
105
106  # calculamos la precision de la prediccion
107  precision = 100 - np.mean(error_medio)
108  print('Precisión media:', round(precision, 2), '%.')
109
```

Observemos los resultados de la predicción de nuestro modelo:

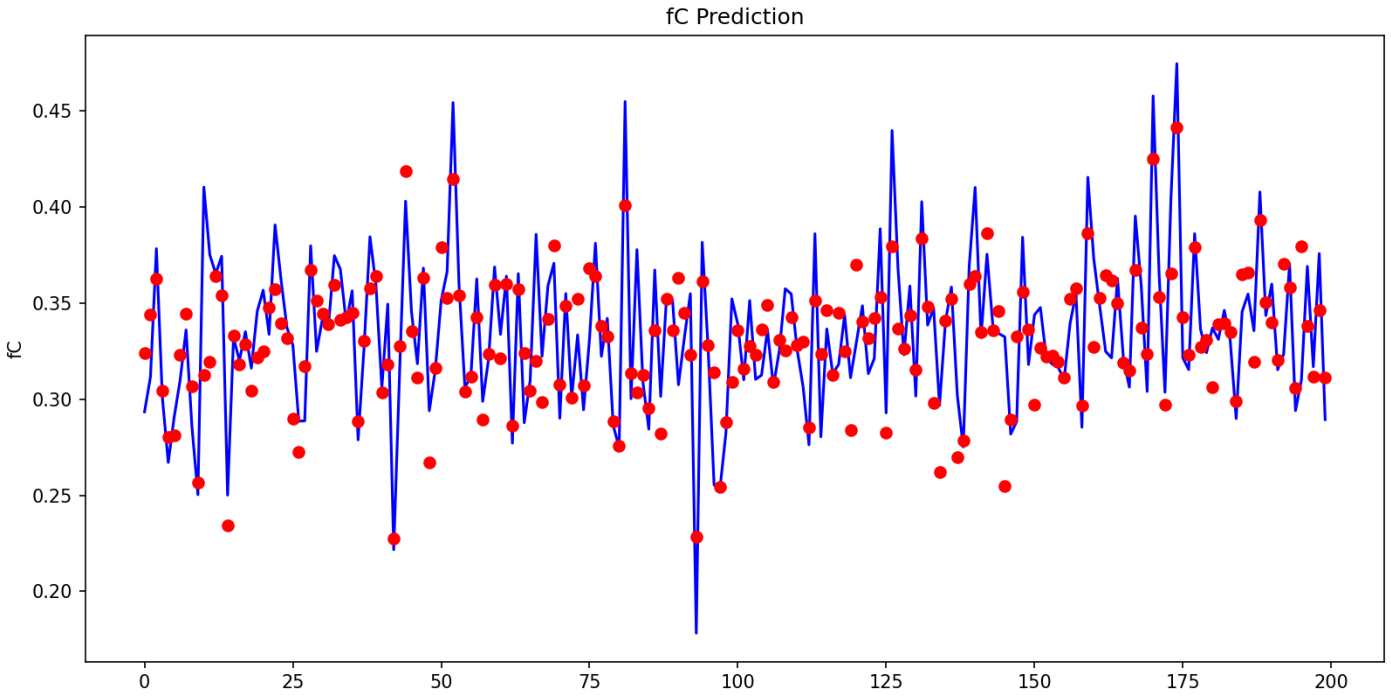
```
1265/1265 [==============================] - 2s 2ms/step
[[0.3237797 ]
 [0.34391695]
 [0.36273766]
 ...
 [0.37356353]
 [0.33151537]
 [0.3052113 ]]
Error absoluto medio: 0.02
Precisión media: 99.68 %.
```

Como podemos ver, la precisión media ha sido de un 99.68%. Esto indica que la fiabilidad de nuestro modelo de regresión es notablemente alta, y que las predicciones que realicen serán muy parecidas a los valores que se obtendrían en la realidad.

Para comprobar a nivel visual la precisión de las predicciones de nuestro modelo de regresión, vamos a generar un gráfico 2D donde plasmemos los valores reales a la par que las predicciones, para ver cómo distan unas de otras. Reduciremos el número de valores a 200, para permitir una visualización más precisa y evitar la acumulación de información.

```
103  # ------------------------------------------------------
104  # visualizacion de resultados
105
106  # añadimos los valores reales y la prediccion al grafico
107  plt.plot(y_test[:200], 'b-', label = 'actual')
108  plt.plot(y_pred[:200], 'ro', label = 'prediction')
109
110  # ajustamos los titulos
111  plt.ylabel('fC')
112  plt.title('fC Prediction')
113
114  # mostramos el grafico
115  plt.show()
```

Observemos la salida del gráfico:



Como podemos observar en el gráfico, la amplia mayoría de las predicciones se encuentran muy cerca del valor real, siendo escasas las predicciones donde el valor predicho diste notablemente del valor real.

Conclusión

Consideramos, por tanto, que el modelo de regresión mediante una red neuronal *keras* es de una fiabilidad muy alta. Además, el tiempo dedicado al entrenamiento, aunque es mayor que el dedicado en Random Forest, no es un tiempo excesivamente largo, pudiendo ser incluso reducido mediante la eliminación de barridas (*epochs*) que, como hemos mencionado previamente, no han sido necesariamente determinantes para asegurar la calidad del modelo.

En el caso de tener que elegir un modelo de regresión u otro, nos decantaríamos por la opción de la red neuronal. A fin de cuentas, es una metodología que se encuentra al alza por su facilidad de uso, su amplia documentación disponible y su alta fiabilidad en el rendimiento.

3. Reglas de asociación.

Para comenzar la extracción de las reglas de asociación vamos a elegir e instalar el paquete de python “apyori”. Este es es una implementación simple del algoritmo Apriori con Python 2.7 y 3.3 - 3.5, proporcionado como API y como interfaces de línea de comandos.

```
!pip install apyori
```

```
Collecting apyori
  Downloading apyori-1.1.2.tar.gz (8.6 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: apyori
  Building wheel for apyori (setup.py) ... done
  Created wheel for apyori: filename=apyori-1.1.2-py3-none-any.whl size=5974 sha256=9c3ec23e3f17e2e05951cf0b9ca3bba8f0a209a55aa
c2bcee18b1e5da5a9a75e
  Stored in directory: /home/jovyan/.cache/pip/wheels/32/2a/54/10c595515f385f3726642b10c60bf788029e8f3a1323e3913a
Successfully built apyori
Installing collected packages: apyori
Successfully installed apyori-1.1.2
```

De esta manera, importamos las librerías que vamos a utilizar. Junto con la librería apyori, vamos a usar la librería pandas.

```
import pandas as pd
from apyori import apriori
```

Descargamos el conjunto de datos proporcionados por este enlace y lo importamos a nuestro código:

https://www.dropbox.com/s/7rfh9hgvwj0nok0/cesta_compra2.csv?dl=0

Usaremos este conjunto de datos para extraer unas reglas de asociación al encontrar relación entre los datos del dataset. Observamos que este archivo presenta un problema:

	0
0	Usuario,Fecha,Item
1	0440,2020-07-14,leche entera,chocolate
2	0440,2020-07-26,bollos,bollos
3	0440,2021-09-06,bolsas de compra,bolsas de compra
4	0440,2020-01-01,otros vegetales,yogurt

Las columnas están separadas con comas. Los datos de la columna ‘Ítem’ están separadas de la misma forma de cómo están separadas. Para solucionar, hacemos una transformación de la lectura para tomar la lista de items en un único campo.

```
df = pd.read_table('cesta_compra2.csv', header=None)
df = df[0].str.split(',', 2, expand=True)
df.columns = df.iloc[0]
df = df[1:]
```

Gracias a esta a transformación, la lista de ítems se puede leer correctamente:

```
df.head()
```

	Usuario	Fecha	Item
1	0440	2020-07-14	leche entera,chocolate
2	0440	2020-07-26	bollos,bollos
3	0440	2021-09-06	bolsas de compra,bolsas de compra
4	0440	2020-01-01	otros vegetales,yogurt
5	0440	2020-01-06	frutas tropicales,zumo de frutas

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14963 entries, 1 to 14963
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   Usuario  14963 non-null  object
 1   Fecha    14963 non-null  object
 2   Item     14963 non-null  object
dtypes: object(3)
memory usage: 350.8+ KB
```

Podemos ver que el conjunto de datos de nuestro dataset hace referencia a listas de compras realizadas por usuarios en una fecha concreta. Cada columna contiene 14963 filas las cuales cada una almacena los ítems que se han comprado junto a los datos de identificación de la propia compra.

Ahora que sabemos el número de filas totales del dataset. Creamos una array para almacenar las listas de los ítems de cada compra. Para ello, hacemos un bucle para ir añadiendo a este array cada uno de los productos de cada compra. Necesitamos tener esta información almacenada en una lista antes de generar las asociaciones.

```
List = []

for k in range(14963):
    List.append(df['Item'][k+1].split(','))
```

Aplicamos el algoritmo a priori para sacar las reglas de asociación sobre el dataset de compras. Para crear este algoritmo, vamos poniendo diferentes parámetros (dando valores

que tengan sentido dentro del significado de nuestro dataset) para intentar conseguir un conjunto de reglas reducidas. Necesitamos introducir cuatro parámetros para filtrar el algoritmo:

1. **List:** Lista de los productos de cada compra del .csv que se utilizará para generar las reglas de asociación a través del algoritmo a priori.
2. **min_support:** como su palabra dice, trata sobre el mínimo soporte que se va usar para extraer las asociaciones de nuestro conjunto de datos. Vamos a elegir un valor bajo porque, dada la diversidad y el tamaño del conjunto de datos, la probabilidad de adquirir un subconjunto de elementos a la vez es muy baja, pero no tan alta como para que el grado los elementos de no son casos aislados.
3. **min_confidence:** como su palabra dice, trata sobre la mínima confianza que se va usar para extraer las asociaciones de nuestro conjunto de datos.Este es un valor bajo, por la misma razón que se explicó anteriormente, con la diferencia de que no hay mucha correlación entre comprar un subconjunto de artículos y otro.
4. **min_lift:** como su palabra dice, trata sobre la mínima confianza que se va usar para extraer las asociaciones de nuestro conjunto de datos. Este es un valor bajo, por la misma razón que se explicó anteriormente, pero mayor que 1 que indica que el grupo ocurre con más frecuencia de lo esperado en modo independiente, y también evita listas vacías.

```
ResultadoApyori = apriori(List, min_support=0.001, min_confidence=0.1, min_lift=1.25)
ListApyori = list(ResultadoApyori)
```

Una vez aplicado el algoritmo a priori y guardado en la variable ResultadoApyori, lo pasamos a formato lista.

Vamos a crear la función ReglasAsociación para crear una tabla con la salida de las reglas de la asociación. Para no generar un número de reglas de asociación demasiado elevado , vamos añadir tres variables a nuestra función que usaremos como filtro en el momento de extraer las asociaciones:

- **Soporte:** Regularidad con la que un conjunto de productos aparece en las distintas transacciones del conjunto de datos**.** Se calcula sumando el número de entradas que contienen X e Y dividido por el número total.
- **Confianza:** Regularidad con la que la regla de asociación es correcta**.** Se calcula viendo con qué frecuencia Y está en las transacciones que X tiene.
- **Lift:** rango de independencia en la manifestación de un conjunto de productos frente a otro. Se calcula a través de la confianza de X+Y entre el soporte de Y.

Las reglas de asociación permitirán saber que productos condicionan a compra de otros y cuales son los habituales.

```
def ReglasAsociacion(ListApyori):

    MotivoCompra = [tuple(ResultadoApyori [2][0][0]) for ResultadoApyori  in ListApyori ]
    RazonCompra = [tuple(ResultadoApyori [2][0][1]) for ResultadoApyori  in ListApyori ]

    Soporte = [ResultadoApyori [1] for ResultadoApyori  in ListApyori]

    Confianza = [ResultadoApyori [2][0][2] for ResultadoApyori  in ListApyori ]
    Lift = [ResultadoApyori [2][0][3] for ResultadoApyori  in ListApyori]

    return list(zip(MotivoCompra, RazonCompra, Soporte, Confianza, Lift))
```

Mostramos las asociaciones generadas por pantalla al transformar la salida a una tabla:

```
pd.DataFrame(ReglasAsociacion(ListApyori), columns = ['| O motivo da compra deste producto ' , 'está condiconada por esta |', 'So
```

```
Soporte', 'Confianza', 'Lift'])
```

	| O motivo da compra deste producto	está condiconada por esta |	Soporte	Confianza	Lift
0	(frutas empaquetadas,)	(bollos,)	0.001203	0.141732	1.288421
1	(productos de temporada,)	(bollos,)	0.001002	0.141509	1.286395
2	(queso gratinar,)	(bollos,)	0.001470	0.144737	1.315734
3	(chicle,)	(yogurt,)	0.001403	0.116667	1.358508
4	(detergente,)	(yogurt,)	0.001069	0.124031	1.444261
5	(harina,)	(frutas tropicales,)	0.001069	0.109589	1.617141
6	(queso gratinar,)	(hortalizas,)	0.001069	0.105263	1.513019
7	(infusiones,)	(yogurt,)	0.001136	0.107595	1.252874
8	(queso freco,)	(yogurt,)	0.001270	0.126667	1.474952
9	(bollos, salchichas)	(leche entera,)	0.001136	0.212500	1.345594
10	(salchichas, leche entera)	(yogurt,)	0.001470	0.164179	1.911760

Esta tabla que hemos creado muestra las asociaciones que pueden servir para que los supermercados analicen donde deben poner los productos en sus locales. De esta manera, pueden ver que productos deben estar cerca unos de otros para conseguir un mayor número de ventas. Hemos conseguido extraer 11 reglas de asociaciones: del 0 al 10.

- **Soporte.** La regla de asociación con un valor más elevado es la número 2.
- **Confianza:** La regla de asociación con un valor más elevado es la número 9
- **Lift:** La regla de asociación con un valor más elevado es la número 10