FOOTBALL ANALYSIS

1. BÚSQUEDA DATASETS

Hasta ahora en el proyecto hemos hecho uso de dos Bases de Datos gratuitas encontradas en Internet:

- Dataset público usado para un estudio en la revista Nature[1]: contiene datos más específicos y concretos de cada partido, incluyendo estadísticas de los jugadores, eventos en el partido, etc.
- Football-Data[2]: una web que tiene publicados datos como estadísticas básicas sobre los partidos y especialmente las cuotas de las casas de apuestas de muchas ligas de todo el mundo.

En el punto en el que estamos actualmente, aún no hemos añadido características muy complejas del Dataset de Nature.

1.1. EXPLORANDO EL NATURE DATASET

En este dataset tenemos incluidos una temporada (2017-18) de las 5 grandes ligas (española, inglesa, italiana, alemana y francesa) y además la Copa del Mundo de 2018 y la Eurocopa de 2016. Sin embargo nos centraremos solamente en las competiciones ligueras ya que son el mismo tipo de competición.

Competition	#matches	#events	#players
Spanish first division	380	628,659	619
English first division	380	643,150	603
Italian first division	380	647,372	686
German first division	306	519,407	537
French first division	380	632,807	629
World cup 2018	64	101,759	736
European cup 2016	51	78,140	552
	1,941	3,251,294	4,299

Figura 1: tabla del contenido del Dataset Nature

[1] https://www.nature.com/articles/s41597-019-0247-7
[2] https://www.football-data.co.uk/data.php

En total haremos uso de 1826 partidos ligueras, que contienen más de 3 millones de eventos y mas de 3000 jugadores.

El dataset dispone de varios elementos que se almacenan en formato JSON. Voy a mostrar cada elemento, explicando algunos atributos importantes o confusos que uso y mostrando algún ejemplo de como se almacenan los datos:

- **Competición**
  - wyId es el identificador propio de este dataset.
  - type indica si la competición es de clubes o de selecciones.


```
{"area": {"alpha2code": "IT",
  "alpha3code": "ITA",
  "id": "380",
  "name": "Italy"},
 "format": "Domestic league",
 "name": "Italian first division",
 "type": "club",
 "wyId": 524}
```


Figura 2: ejemplo archivo JSON de competición

- **Partidos (Matches)**
  - teamsData contiene muchos subcampos que describen la información de cada equipo que juega el partido, por ejemplo si tiene formación, los goles marcados en el partido, si juega de local o visitante.

    Los equipos se identifican por el identificador de la DB.

  - En el subcampo de Formation (si hasFormation = 1) tenemos tanto los jugadores titulares, como los sustitutos y el resto banquillo.

```
{"competitionId": 524,
 "date": "May 20, 2018 at 8:45:00 PM GMT+2",
 "dateutc": "2018-05-20 18:45:00",
 "duration": "Regular",
 "gameweek": 38,
 "label": "Lazio - Internazionale, 2 - 3",
 "roundId": 4406278,
 "seasonId": 181248,
 "status": "Played",
 "venue": "Stadio Olimpico",
 "winner": 3161,
 "wyId": 2576335,
 "referees": [{"refereeId": 377206, "role": "referee"}
    , {"refereeId": 384945, "role": "secondAssistant"}
    , {"refereeId": 377241, "role": "fourthOfficial"}]
    ,

 "teamsData":

 {"3161": {"coachId": 101547,
   "formation": {"bench": [...],
    "lineup": [...],
    "substitutions": [...]},
   "hasFormation": 1,
   "score": 3,
   "scoreET": 0,
   "scoreHT": 1,
   "scoreP": 0,
   "side": "away",
   "teamId": 3161},

  "3162": {"coachId": 251025,
   "formation": {"bench": [...],
    "lineup": [...],
    "substitutions": [...]},
   "hasFormation": 1,
   "score": 2,
   "scoreET": 0,
   "scoreHT": 2,
   "scoreP": 0,
   "side": "home",
   "teamId": 3162}}}
```


Figura 3: ejemplo archivo JSON de un partido

```
"formation":
{"bench":
[{"goals": "0", "ownGoals": "0", "playerId": 3543, "
   redCards": "0", "yellowCards": "0"},
      ...
{"goals": "0", "ownGoals": "0", "playerId": 20519, "
   redCards": "0", "yellowCards": "0"}],
"lineup":
[{"goals": "0", "ownGoals": "0", "playerId": 20556, "
   redCards": "0", "yellowCards": "0"},
         ...
{"goals": "0", "ownGoals": "0", "playerId": 135903, "
   redCards": "0", 'yellowCards': "0"}],
"substitutions":
[{"minute": 60, "playerIn": 20626, "playerOut": 20556}
   ,
{"minute": 67, "playerIn": 352993, "playerOut": 3344},
         ...
]}}
```


Figura 4: ejemplo archivo JSON del campo formación de un equipo en un partido

- **Equipos** (Teams)


```
{"city": "Torino",
 "name": "Juventus",
 "area": {"alpha2code": "IT", "name": "Italy", "alpha3
    code": "ITA", "id": "380"},
 "wyId": 3159,
 "officialName": "Juventus FC",
 "type": "club"}
```


Figura 5: ejemplo archivo JSON del equipo Juventus

- **Evento**

  Como se muestra en la Figura 7, los eventos se clasifican en tipos generales (pases, faltas, disparos...), los cuales tienen su propio ID (eventId).

  Luego se clasifican con más detalle con un subtipo (p.e. en pases estan los centros y los pases simples), estos tambien tienen su propio ID (subEventId).

  Finalmente, cada tipo de evento tiene unos tags que añaden información del evento.

  - id identificador de cada evento
  - eventId el identificador del tipo de evento
  - eventName nombre del tipo de evento
  - subEventId y subEventName
  - eventSec tiempo de la parte en el cual ocurre el evento

- matchPeriod parte del partido en el que ocurre (1H/2H,E1,P...)

type	subtype	tags
pass	cross, simple pass	accurate, not accurate, key pass, opportunity, assist, goal
foul		no card, yellow, red, 2nd yellow
shot		accurate, not accurate, block, opportunity, assist, goal
duel	air duel, dribbles, tackles, ground loose ball	accurate, not accurate
free kick	corner, shot, goal kick, throw in, penalty, simple kick	accurate, not accurate, key pass, opportunity, assist, goal
offside touch	acceleration, clearance, simple touch	counter attack, dangerous ball lost, missed ball, interception, opportunity, assist, goal

Figura 6: tipos y subtipos de eventos


```
{"eventId": 8,
 "eventName": "Pass",
 "eventSec": 2.4175,
 "id": 253668302,
 "matchId": 2576335,
 "matchPeriod": "1H",
 "playerId": 3344,
 "positions":
 [{"x": 49, "y": 50}, {"x": 38, "y": 58}],
 "subEventId": 85,
 "subEventName": "Simple pass",
 "tags":
 [{"id": 1801}],
 "teamId": 3161}
```


Figura 7: JSON evento

- **Jugador** (Player) (Figura 6)
  - role la posición, el rol que juega en el campo


```
{"birthArea":
{"alpha2code": "AR",
  "alpha3code": "ARG",
  "id": "32",
  "name": "Argentina"},
 "birthDate": "1987-06-24",
 "currentNationalTeamId": 12274,
 "currentTeamId": 676,
 "firstName": "Lionel Andres",
 "foot": "left",
 "height": 170,
 "lastName": "Messi Cuccittini",
 "middleName": "",
 "passportArea": {"alpha2code": "ES",
  "alpha3code": "ESP",
  "id": "724",
  "name": "Spain"},
 "role": {"code2": "FW", "code3": "FWD", "name": "
    Forward"},
 "shortName2: "L. Messi",
 "weight": 72,
 "wyId": 3359}
```

Figura 8: ejemplo archivo JSON del jugador Lionel Messi

1.2. EXPLORANDO FOOTBALL-DATA

Cada temporada de cada liga tiene su propio archivo CSV que siguen el mismo patrón de nombre: PaísDivision_Temporada, siendo País las siglas del país, p.e. SP para España; Division si se trata de 1ª seria 1; y Temporada p.e. 1516 para la 2015-16; el nombre del fichero de este ejemplo sería SP1_1516.csv.

Es importante destacar que las características, es decir las columnas, de cada país varían un poco ya que las casas de apuestas cambian de un país a otro. Sin embargo nosotros vamos a ocuparnos de momento en una casa de apuestas que está presente en todas las ligas: BET365.

Un ejemplo de dataset de Football-Data es el de la siguiente figura:

Div	Date	HomeTeam	AwayTeam	FTHG	FTAG	FTR	HTHG	HTAG	HTR	...	AF	HC	AC	HY	AY	HR	AR	B365H	B365D	B365A
SP1	18/08/17	Leganes	Alaves	1	0	H	1	0	H	...	18	4	2	0	1	0	0	2.05	3.20	4.10
SP1	18/08/17	Valencia	Las Palmas	1	0	H	1	0	H	...	13	5	2	3	3	0	1	1.75	3.80	4.50
SP1	19/08/17	Celta	Sociedad	2	3	A	1	1	D	...	11	5	4	3	1	0	0	2.38	3.25	3.20
SP1	19/08/17	Girona	Ath Madrid	2	2	D	2	0	H	...	15	6	0	2	4	0	1	8.00	4.33	1.45
SP1	19/08/17	Sevilla	Espanol	1	1	D	1	1	D	...	12	7	3	2	4	1	0	1.62	4.00	5.50

Figura 9: ejemplo Football-Data liga española 2017-18

Realmente hay muchas más características en el dataset pero aquí muestro sobretodo las que hemos usado y nos parecen más importante.

Tenemos características fundamentales de los partidos como **FTHG, FTAG** o **FTR** que nos indican los goles anotados por el Local, Visitante y quien ha ganado respectivamente, si el local (H), el visitante (A) o ha acabado empate (D). También tenemos los goles al descanso (**HTHG** y **HTAG**).

A parte tenemos también otras características muy interesantes, como las faltas cometidas por los equipos (**HF** y **AF**), los disparos (**HS** y **AS),** disparos a puerta, al palo, corners, fueras de juego, tarjetas,etc.

Estas características aparecen en todas las ligas y temporadas que he empleado hasta ahora. Para ver con más detalle la definición de cada característica recomiendo leer el archivo *notes.txt*[3] que está disponible para descargar en la web del dataset.

[3] https://www.football-data.co.uk/notes.txt

Las siguientes características que aparecen son las de las cuotas de las Casas de Apuestas, y como he dicho anteriormente estas casas pueden ir cambiando dependiendo el país, aunque algunas estan presentes en todas o la mayoria de ligas y temporadas. Ese es el caso de BET365, y por ello he utilizado sus cuotas (**B365H** y **B365A**).

Es importante remarcar que no solo tenemos disponibles cuotas simples, sino que disponemos de diferentes tipos de cuotas, como la del Handicap Asiático, cuotas a goles marcados, etc.

2. CREACIÓN DATASETS

Los datasets que creemos serán, por lo general, fusiones de diversos datasets.

2.1. CREANDO NATURE DATASET

Hasta ahora, hemos usado el Dataset de Nature como base para futuros Datasets más grandes y completos. Pero para ello había de tratar y fusionar los distintos JSON para poder manejar los datos en un DataFrame de Pandas.

Por ello, la primera base de datos que he implementado ha sido para unir y organizar los JSON que almacenan los partidos dependiendo de la liga.

Este ha sido el proceso:

Importamos las librerías que vamos a utilizar en la creación de los datasets.

Estas librerías las usaremos en la creación de todos los datasets.

```
import os
import pandas as pd
import numpy as np
import json
import difflib as dl
```

Importamos los JSONs de los partidos y los almacenamos en un diccionario donde las claves son las competiciones.

```
# Loading the match data
matches={}
nations = ['Italy','England','Germany','France','Spain']
for nation in nations:
    with open('./matches/matches_%s.json' %nation) as json_data:
        matches[nation] = json.load(json_data)
```

Concatenamos todos los partidos en un array plano. El resultado será un array de JSONs, donde cada JSON es un partido. Fijarse que solo seleccionamos los partidos de las Ligas.

```
partidos = []

for pais in nations:
```

```
    for partido in matches[pais]:
        partidos.append(partido)

partidos = np.array(partidos,dtype=dict)
```

Para transformar ese array de JSONs a una tabla (dataFrame) donde podamos ver cada partido con sus atributos, transformamos cada JSON a Series (un tipo de objeto Pandas) y almacenamos los partidos (Series) en un array *match_serie*.

Luego cogemos los atributos de un partido cualquiera (en formato Series) y creamos el DataFrame con los atributos del partido.

Al final, hemos seleccionado unos pocos atributos generales de cada partido con los que nos interesa quedarnos de momento, *raw_db*.

```
match_serie = [pd.Series(p) for p in partidos]
rawDB = pd.DataFrame(match_serie, columns=(match_serie[0]).index)

columnNamesDB = ['wyId','competitionId','seasonId','roundId','dateutc','winner']
# wyId = ID del partido en el DB
rawDB = rawDB[columnNamesDB]
```

A continuación añadimos a nuestro *rawDB* los datos de cada equipo que disputa el partido.

Para añadirlos hay que tener en cuenta que los datos de cada equipo se encuentran como un objeto JSON dentro del JSON del partido, es decir dentro del atributo *teamsData*.

Dentro del atributo *teamsData* encontramos dos JSON, uno para cada equipo. El índice de cada JSON es el Id del equipo

```
# Cogemos cada equipo (JSON) de cada partido
teamsData = [list(m['teamsData'].values()) for m in match_serie]
# partido x equipo
# Creamos las listas donde irán los equipos Locales y Visitantes
homeT = []
awayT = []

for m in teamsData:
    if m[0]['side'] == 'home':
        homeT.append(pd.Series(m[0]))
        awayT.append(pd.Series(m[1]))
    else:
        homeT.append(pd.Series(m[1]))
        awayT.append(pd.Series(m[0]))
```

```
# debemos cambiar el nombre de las columnas para que no se llamen igual las Home
que las Away
colH = [str(x) + '_home' for x in homeT[0].index]
colA = [str(x) + '_away' for x in awayT[0].index]
# Creamos dos DataFrames, cada uno tendrá los atributos de cada equipos H y A
hData = pd.DataFrame(homeT, columns=homeT[0].index)
aData = pd.DataFrame(awayT, columns=awayT[0].index)
hData.columns = colH
aData.columns = colA

assert len(hData) == len(aData) == len(rawDB)

# Concatenamos estos dos DataFrames que acabamos de crear al que ya habiamos
creado
rawDB = pd.concat([rawDB,hData,aData],axis=1)
```

En este momento en *rawDB* tendremos los siguientes atributos:

```
Index(['wyId', 'competitionId', 'seasonId', 'roundId', 'dateutc', 'winner',
       'scoreET_home', 'coachId_home', 'side_home', 'teamId_home',
       'score_home', 'scoreP_home', 'hasFormation_home', 'formation_home',
       'scoreHT_home', 'scoreET_away', 'coachId_away', 'side_away',
       'teamId_away', 'score_away', 'scoreP_away', 'hasFormation_away',
       'formation_away', 'scoreHT_away'],
      dtype='object')
```

Figura 10: todas las columnas de rawDB

Todos los atributos que tenemos ahora son Integers, Strings, Dates, Booleanos salvo *formation_home/away* que contiene un JSON con la formacion del equipo, como hemos visto en el apartado que explicaba el dataset de Nature.

Por último nos quedamos solamente con varios atributos, para crear los siguientes Datasets. Realizamos también varias comprobaciones.

```
rawDB = rawDB[[
    'wyId', 'competitionId', 'seasonId', 'roundId', 'dateutc', 'winner',
    'teamId_home','score_home','scoreHT_home','teamId_away', 'score_away',
    'scoreHT_away'
     ]]
# comprobamos que estan todos los partidos y que no hay ninguno repetido
assert len(rawDB) == 1826 == len(set(rawDB['wyId']))
```

2.2. AÑADIMOS CUOTAS AL DATASET

Después de transformar nuestro Dataset de JSONs de Nature a Pandas DataFrames, le añadimos las cuotas de *Football-Data*.

Vamos a añadir las cuotas a nuestro Dataset porque tener como base las predicciones de los partidos que hacen las grandes Casas de Apuestas nos puede servir como modelo base con el que comparar futuros modelos.

Para ello importamos los CSV de la temporada 2017-18 de las 5 ligas, concatenaremos los CSV en un mismo DataFrame y nos quedaremos solamente con los siguientes atributos:

```
['Div','Date','HomeTeam','AwayTeam','FTHG', 'FTAG', 'FTR','B365H', 'B365D',
'B365A']
```

Ahora, el gran problema es identificar los partidos en ambas bases de datos. Es decir, tenemos que encontrar la fórmula para identificar cada partido en ambas BD.

Para ello he usado la libreria *difflib*, que com su metodo *get_close_matches(String,list)* que devuelve en una lista los elementos de la *list* que más se parecen al *String*.

Para encontrar la solución al problema he ido por pasos:

Antes que nada, guardamos en 2 listas los nombres de los equipos en cada BD.

El primer paso es buscar que equipos tienen el mismo nombre en ambas BD, y guardamos esos equipos en una lista llamada *iguales*, y los demás en *desiguales*. Hay que puntualizar que guardamos en esas listas los nombres de Nature, los de Football-Data los usamos para comparar.

El siguiente paso es encontrar dentro de los *desiguales* los nombres semejantes en Football-Data. Para ello usamos el metodo de la libreria *difflib*

```
encontrados = []
no_encontrados = []

for des in desiguales:
```

```
    find = dl.get_close_matches(des,fdTeams)
# fdTeams = nombres de Football-Data
    if len(find)>0:
        encontrados.append((des,find[0]))
    else:
        no_encontrados.append(des)
```

Como resultado tenemos en *encontrados* los nombres que ha visto similares en ambos BD. Los guardo como pares *(nombre_Nature, nombre_FD)*.

Los restantes, *no_encontrados*, deberemos cambiarlos a mano (en total 12 equipos de 98).

Finalmente unimos todos en un diccionario *equipos* de esta forma:

```
equipos = {} # { nature_data : football_data } }
```

Ahora, para poder añadir correctamente las cuotas a cada partido en la BD de Nature, tenemos que tener un diccionario con el cual las claves sean los IDs de los equipos y los valores sean el nombre del equipo, debido a que en los partidos se identifica al equipo por el ID y no por el nombre. El diccionario se crea facilmente iterando sobre el JSON de *teams*.

Un vez hecho esto, iteramos sobre cada partido de Nature y sobre cada partido de Football-Data, buscando la correspondencia de IDs y nombres, tanto para el local como para el visitante en cada partido.

El código de las iteraciones sería tal que así

```
for h,a in zip(rawDB['teamId_home'],rawDB['teamId_away']):
# rawDB = DB de Nature
    for i in range(1826):
        p = fdataDB.loc[i] # fdataDB = DB de Football-Data
        # nat_ID_Names = { ID : nombre_equipo } -> equivalencia ID - nombreNAT
        # equipos -> equivalencia nombreNAT -> nombreFD
        h_name = equipos[nat_ID_Names[h]]
        a_name = equipos[nat_ID_Names[a]]
        if p['HomeTeam']==h_name and p['AwayTeam']==a_name:
            bet_h_nat.append(p['B365H'])
            bet_a_nat.append(p['B365A'])
            break
```

Esta es una de las partes más costosas computacionalmente de la creación de los datasets.

Por último exportamos el Dataset a CSV, con nombre *cuotaNatDB*.

2.3. TENDENCIAS ÚLTIMOS N PARTIDOS

En la segunda versión del Dataset de Cuotas, hemos decidido crear una nueva característica que he pensado que puede aportar nueva información a nuestro modelo con el fin de mejorar su eficacia.

Esta nueva característica se basa en el rendimiento del equipo en los últimos *n* partidos que ha disputado. En concreto, se basa en los puntos que ha conseguido en esos últimos *n* partidos.

La formula seria la siguiente:

$$tendencia(e) \ = 1/N \ \bullet \ \sum_{i=1}^{N} p_{e\,i}$$

siendo $p \in \{0, 1, 3\}$.

Esto nos daría dos características, dos columnas independientes entre sí que serían **tendencia_H** y **tendencia_A**.

Probamos con N=5, y exportamos el Dataset a CSV, con nombre *cuotasDB_tend5.*

En un futuro, podriamos expandir esta misma idea para los goles marcados, las faltas o la formación, al igual que con un valor de N distinto.

Otra versión

Otra manera de representar esta información en nuestro dataset de una manera más compacta sería reduciendo ambas columnas a una sola.

La idea seria restar una a la otra y dependiendo el signo del resultado un equipo llegaría al partido con mejor tendencia de resultados que su contrincante.

El nuevo atributo **dif_tend** se calcula de la siguiente manera:

$$diftend \ = tendenciaA \ - \ tendenciaH$$

De esta manera si dif_tend < 0, el local llegaria al partido con mejores resultados en los últimos N partidos que el visitante, y si dif_tend > 0 lo

contrario. En caso de dif_trend = 0 significaria que ambos llegan con idénticos puntos en los últimos N partidos.

3. IMPLANTACIÓN MODELOS PREDICTIVOS

3.1. PREPROCESAMIENTO DE LOS DATOS

Antes de implementar sobre nuestros datos algún modelo predictivo, realizamos algunas modificaciones y transformaciones sobre los datos de los datasets explicados en los capítulos anteriores.

Estas modificaciones y transformaciones servirán para que nuestros datos "en bruto" estén representados de una manera más entendible y eficaz por los modelos que les apliquemos.

Para implementar las funciones de transformación he creado una función en Python, la cuál se encarga de entrenar los datos con la función de preprocesamiento que se le pase.

```
# funcion para normalizar/estandarizar DB

def normalDB(db,scaler='', col_res='res'):
    if scaler=='':
        from sklearn.preprocessing import MinMaxScaler
        scaler = MinMaxScaler()

    res  = db[col_res]
    data = db.drop(col_res,axis=1)
    cols = data.columns

    scaler.fit(data)

    data = scaler.transform(data)
    data = pd.DataFrame(data,columns=cols)
    data[col_res] = res

    return data
```

Figura 10: funcion de estandarización/normlizacion

A esta función le pasamos como parámetros, de forma obligatoria el dataset, y de forma opcional tanto una clase *scaler* de la librería sklearn.preprocessing, como *col_res* donde indicaremos cuál es la columna de las etiquetas de nuestras muestras, ya que la etiqueta no nos interesa que se normalice o estandarice.

El *scaler* se encargará de estandarizar o normalizar nuestros datos dependiendo de qué clase de escalador le pasemos. Por defecto, si no le pasamos ninguno, entrenará el *MinMaxScaler* en nuestro dataset.

Destacar, que en el caso de las cuotas, al ser el inverso de la probabilidad y tener un valor mayor que 1, simplemente volviendo a invertirlas tenemos la probabilidad (de 0 a 1) de que ocurra un hecho (en nuestro caso ganar local, empate o ganar visitante). Es por eso que en este caso excepcional no sé aprecian resultados significativamente diferentes si después de invertir las cuotas aplicamos una función de estandarización o no.

	B365H	B365A	res	tendencia_H	tendencia_A
0	2.40	2.90	1	2.2	1.8
1	4.00	1.85	1	2.4	2.6
2	3.10	2.25	-1	0.8	2.2
3	1.53	5.75	-1	1.4	1.4
4	1.53	6.00	-1	0.8	0.2
...	...	...	...	...	...
1821	1.62	5.50	0	0.0	0.0
1822	8.00	1.45	0	0.0	0.0
1823	2.38	3.20	1	0.0	0.0
1824	1.75	4.50	-1	0.0	0.0
1825	2.05	4.10	-1	0.0	0.0

1826 rows × 5 columns

→

	B365H	B365A	tendencia_H	tendencia_A	res
0	0.403808	0.368673	0.733333	0.600000	1
1	0.221443	0.597431	0.800000	0.866667	1
2	0.300860	0.485110	0.266667	0.733333	-1
3	0.663052	0.168900	0.466667	0.466667	-1
4	0.663052	0.160430	0.266667	0.066667	-1
...	...	...	...	...	...
1821	0.623321	0.178140	0.000000	0.000000	0
1822	0.084669	0.771724	0.000000	0.000000	0
1823	0.407639	0.330887	0.000000	0.000000	1
1824	0.573146	0.225366	0.000000	0.000000	-1
1825	0.481646	0.250707	0.000000	0.000000	-1

1826 rows × 5 columns

Figura 11: dataset antes y después de aplicar inversión de la cuota y *MinMaxScaler*(0,1)

Tras aplicar esta primera transformación, debemos crear nuestros datasets de entrenamiento y test. Para ello hemos creado otra función.

```
def preprocessDB(db,frac=0.85,col_res='res',seed=1):
    db = db.sample(frac=1,axis=0,random_state=seed)
    index = int(len(db)*frac)
    y = db[col_res]
    X = db.drop(col_res,axis=1)
    trainX = X[:index]
    testX  = X[index:]
    trainY = y[:index]
    testY  = y[index:]

    trainDB = db[:index]

    return trainDB,trainX,trainY,testX,testY
```

Figura 12: funcion de preprocesamiento

A esta función le pasamos como parámetros, de forma obligatoria el dataset, y opcionalmente *frac* (% destinado al entrenamiento), *col_res* (el atributo que representa la etiqueta del objeto, el label), *seed* (la semilla para randomizar los datos de una manera reproducible).

Una vez pasados los parámetros, randomizamos todos los datos con la función de pandas *sample*, a partir de la semilla y del eje 0, es decir de las filas. A continuación calculamos el numero de muestras de entrenamiento, guardamos en una lista las etiquetas (que no seran transformadas), eliminamos, por tanto, esta columna del dataset. Por último creamos las variables que almacenaran por separado las muestras de entrenamiento (*trainX*), las etiquetas de estas muestras (*trainY*), las muestras de test (*testX*), las etiquetas de test (*testY*) y finalmente una variable que contendrá todos los datos de entrenamiento y que será usada para mostrar los gráficos por ejemplo.

En un capitulo próximo mostraremos el resultado de usar diferentes *scalers*, así como diferentes tamaños de datasets de entrenamiento.

3.2. VISUALIZACIÓN DATASETS

Antes de probar cualquier modelo es muy importante saber que datos tienes, como estan distribuidos y que atributos interesan. Para ello, la visualización de los datos nos puede dar pistas de cómo va a funcionar un modelo.

Para la visualización en gráficas, voy a usar principalmente la libreria de Python *seaborn*. A continuación, mostraré y comentaré los resultados de nuestros datasets.

Cuotas

Recordemos que el dataset de *CuotasDB* era el dataset más básico que tenemos, con sólo dos atributos, las cuotas de BET365 de local y de visitante.

Tras aplicarle las transformaciones que hemos explicado antes (usando MinMaxScaler y creando los datasets de entrenamiento y test), vamos a visualizar como se distribuyen sus datos.

El primer gráfico representa como se distribuyen los distintos resultados (victoria local (**-1**), victoria visitante (**1**) y empate (**0**)) según la cuotas que predicen las casas de apuestas.

Se ve representada también, con una linea azul, la función $y = 1 - x$.

El propósito de representar la línea azul, es para mostrar como las Casas de Apuestas dejan un margen para el empate, y también para asegurar beneficio.

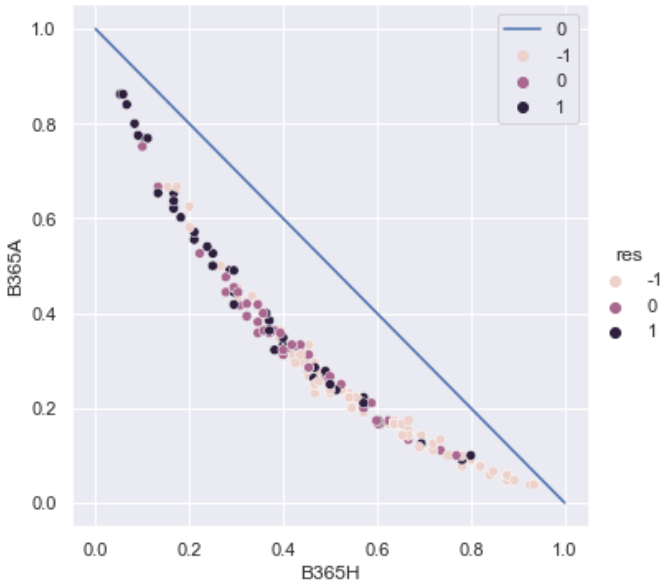

Figura 13: distribución datos CuotasDB

Resumidamente, la distribución es bastante lógica ya que como se puede observar cuanto más alto es el valor de B365H, es decir cuando más probabilidades a priori hay de que gane el local, más muestras etiquetadas con **-1** hay. Y lo mismo para B365A y las victorias visitantes.

Sin embargo, la mayoria de datos se concentran en el medio de la curva, y es en esa zona donde las etiquetas son más heterogéneas y más complicadas de predecir. En las siguientes figuras se aprecia.

Como último apunte, se puede apreciar una cierta dependencia entre ambos atributos (B365H y B365A), en concreto una proporcionalidad inversa. Por lo que se podria considerar en un futuro descartar uno de los dos atributos.

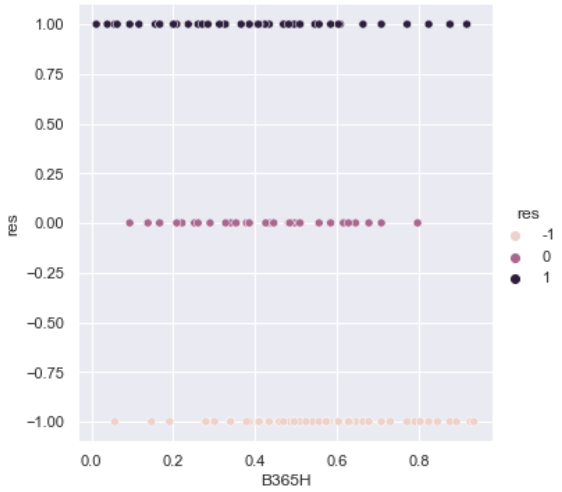

Figura 14: distribución datos B365H

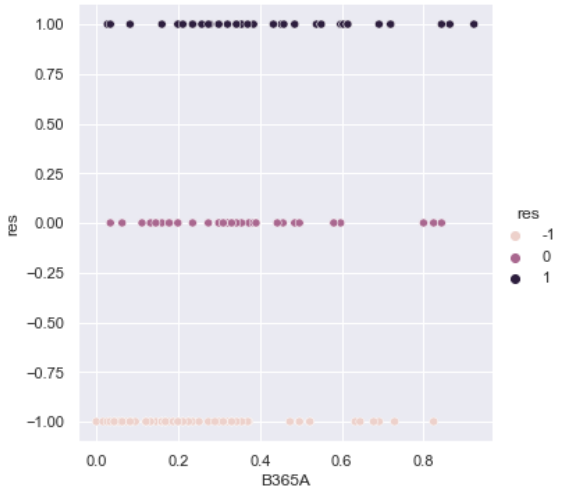

Figura 15: distribución datos B365A

En las Figuras 14 y 15 vemos la distribución de las etiquetas dependiendo solamente de un atributo. Vemos mejor aquí como la gran mayoria de los datos se concentran de forma heterogenea en un segmento, mientras que los extremos, que son más homogéneos contienen un numero marginal de muestras.

Tendencia

A continuación, vamos a mostrar como se distribuyen las etiquetas de las muestras según la diferencia de las tendencias de ambos equipos. Vamos a hacer uso del nuevo atributo creado *dif_tend* explicado en este apartado.

En estos gráficos se muestra la distribución de las tres clases (etiquetas) según la probabilidad de victoria del local y de las tendencias de ambos equipos.

Como podemos observar, la gran mayoria de datos se encuentran en el centro de las gráficas. Añadiendo las tendencias si que tenemos un poco más de información, sin embargo, es pronto para sacar una conclusión sobre si es un atributo que proporcione suficiente información nueva o es totalmente dependiente de otros atributos como las cuotas, algo que podria ser así en el caso de que las Casas de Apuestas tuviesen en cuenta la tendencia de los equipos para decidir las cuotas.

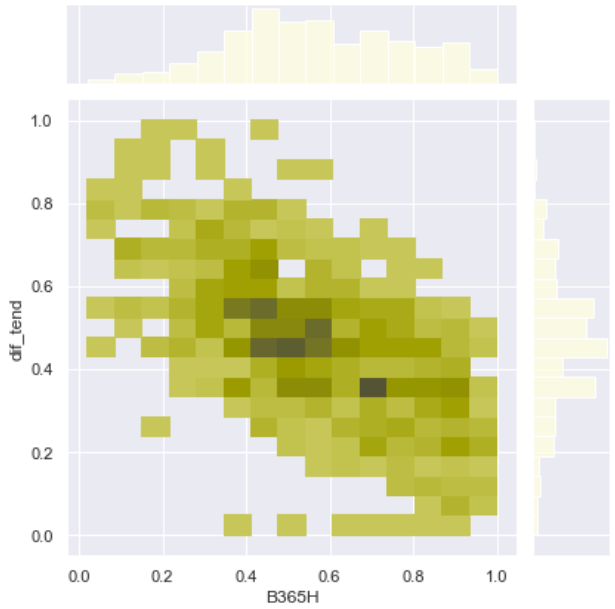

Figura 16: distribución victorias locales según B365H y dif_tend

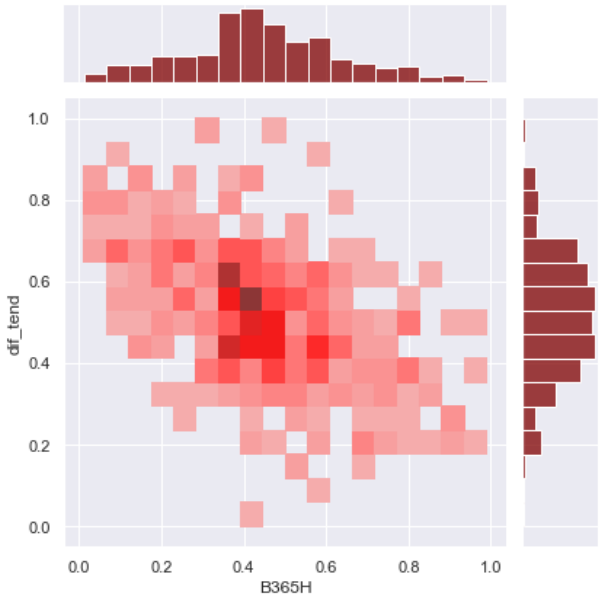

Figura 17:distribución empates según B365H y dif_tend

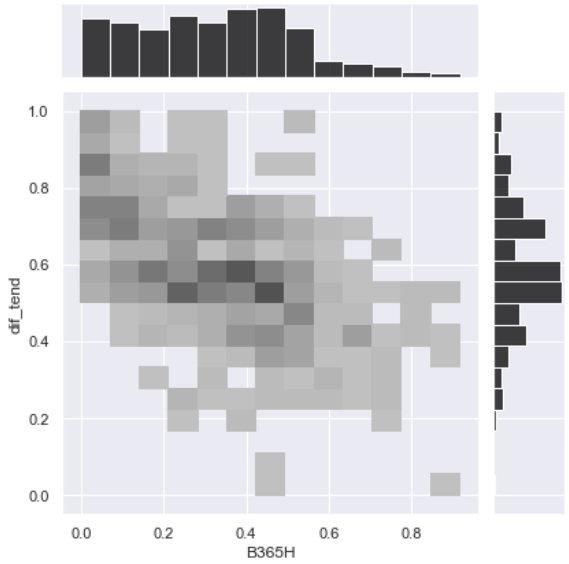

Figura 18:distribución victorias visitantes según B365H y dif_tend

3.3. MODELOS DE PREDICCIÓN

Vamos a implementar un par de modelos muy simples en los datasets creados y visualizados anteriormente.

Primero, veremos por encima explicado como se han implementado y luego evaluaremos los resultados.

Todos los modelos el entrenamiento lo haremos con Validación Cruzada para minimizar el sobreentrenamiento y realizaremos una Búsqueda por Rejilla para encontrar los hiperparámetros más óptimos de los modelos .

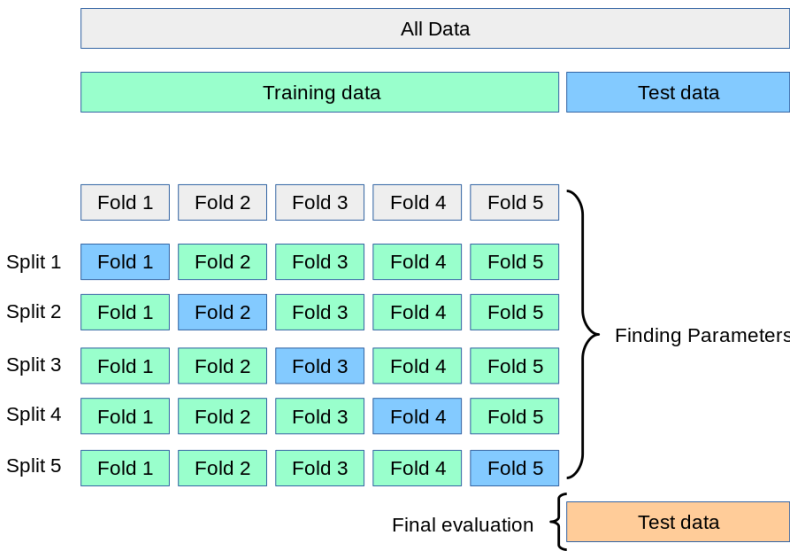

Figura 19: esquema del funcionamiento de la Validación Cruzada

Como se expone en la Figura 19, la Validación Cruzada consiste en dividir los datos de entrenamiento en *n* segmentos iguales, *folds*, para luego hace *n* iteraciones. En cada iteración una *fold* se utilizará como test del entrenamiento, por lo que no se usarán esos datos como muestras de entrenamiento. Con ello se espera que el entrenamiento y la validación del modelo no se realice exactamente con los mismos datos, evitando así el sobreentrenamiento.

Arbol de decisión

El primer modelo que vamos a implementar es un árbol de decisión. Primero vamos a mostrar una primera versión del árbol entrenado solo con Validación Cruzada, es decir, no han sido optimizados los parámetros del árbol por búsqueda por rejilla.

```
from sklearn import tree
from sklearn.model_selection import cross_val_score, cross_val_predict,
cross_validate

clf = tree.DecisionTreeClassifier()
results = cross_validate(clf,trainX,trainY,cv=5,return_estimator=True)
```

Figura 20: lineas de código para entrenar un *DecissionTree* con Validacion Cruzada

A la función *cross_validate* se le pasa como parámetros el modelo a entrenar, los datos de entrenamiento (muestras y etiquetas), el numero de *folds* de la Validación cruzada y para que devuelve el modelo resultante activamos la opción de *return_estimation* a True.

```
In [72]: results

Out[72]: {'fit_time': array([0.00598192, 0.00299406, 0.00298762, 0.00299191, 0.00302362]),
          'score_time': array([0.00202751, 0.00199676, 0.00199461, 0.00196362, 0.00199389]),
          'estimator': [DecisionTreeClassifier(),
           DecisionTreeClassifier(),
           DecisionTreeClassifier(),
           DecisionTreeClassifier(),
           DecisionTreeClassifier()],
          'test_score': array([0.50482315, 0.46451613, 0.48387097, 0.49677419, 0.48064516])}
```

Figura 21: output de la cross_validate

Este es un ejemplo del objeto que devuelve la función *cross_validate*. Destacar especialmente los campos de *estimator* donde podemos ver todos los modelos entrenados y el *test_score* donde obtenemos la puntuación en cada iteración del CV.

El siguiente paso es entrenar el modelo teniendo en cuenta la importancia de optimizar sus hiperparámetros, para ello haremos uso de la Búsqueda por Rejilla (*GridSearch*).

Hay dos formas de aplicar *Grid Search*, con validación cruzada anidada o sin anidar. La diferencia es que si no anidamos la Validación cruzada lo que estamos haciendo es entrenar los hiperparámetros y luego validarlos con los mismos datos que los has entrado; en cambio, si anidamos la Validación Cruzada lo que conseguiremos es validar el entrenamiento con unos datos diferentes a los de entrenamiento. A continuación mostraremos el funcionamiento de ambas y más tarde los resultados para sacar conclusiones.

```
from sklearn.model_selection import GridSearchCV
```

Una vez importada la clase de sklearn.model_selection, *GridSearchCV*, deberemos crear un objeto que sea un clasificador *GridSearchCV* para ello le debemos pasar de parámetros el clasificador base, en nuestro caso un Árbol de Decisión, un diccionario con los hiperparámetros del Árbol y sus valores posibles a optimizar.

```
param = {
    'criterion':["gini","entropy"],
    'splitter':["best","random"],
    'random_state':[None,1],
}

# añadir min_samples_leaf y otros param de hojas
```

Figura 22: hiperparámetros del árbol de decisión a optimizar

Para no anidar la Validación Cruzada, es decir validando el modelo entrenado con las mismas muestras entrenadas, el código es el siguiente:

```
# Sin Anidar
clf_grid = GridSearchCV(estimator=clf, param_grid=param, cv=5)
clf_grid.fit(trainX,trainY)

clf_grid.cv_results_

clf_grid.best_params_
```

Figura 23: entrenamiento GridSearch sin anidacion de Validacion Cruzada

Un ejemplo del objeto que devuelve *GridSearchCV* como resultado es el de la Figura 24. Destacamos los parámetros utilizados en cada iteración de CV, *params*, y a continuación las puntuaciones con cada *split* de CV, *splitX_test_score*. El atributo *std_test_score* nos proporciona la desviación típica del resultado en cada *split.*

El atributo *best_params_* de la clase *GridSearchCV* nos dirá cuales son los parámetros que mejor resultado han dado.

Sin embargo, para entrenar realmente los hiperparámetros nos interesa que la Búsqueda por rejilla se haga anidando la Validación Cruzada, así en cada separación (*split*) se entrenarán estos hiperparámetros con un fragmento diferente de datos y se validarán con el resto.

El código es muy sencillo:

```
clf_grid = GridSearchCV(estimator=clf, param_grid=param, cv=5)
score = cross_validate(clf_grid,X=trainX,y=trainY, cv=5, return_estimator=True)
```

El resultado será muy similar a la Figura 21, pero el *estimator* será una lista de los estimadores *GridSearchCV* entrenados. Podemos acceder a cada estimador a través de esta lista y ver los resultados con más detalle. El atributo resultado *test_score*, muestra el resultado en cada *split*, sin embargo no queda claro en la documentación si es la media o el máximo, ya que en este caso por ejemplo en cada *split* se ejecuta también la validación cruzada en el *GridSearch*. Lo más probable que sea la puntuación de la validación con las muestras de test del CrossValidation usando los mejores hiperparámetros del modelo.

```
clf_grid.cv_results_
```

```
]: {'mean_fit_time': array([0.00478067, 0.00279241, 0.00339756, 0.00299091, 0.00398049,
        0.0027863 , 0.00379152, 0.00299878]),
 'std_fit_time': array([2.63265586e-03, 3.98779044e-04, 4.83406698e-04, 4.02751015e-06,
        1.16250513e-05, 3.97153790e-04, 3.99208781e-04, 1.16483032e-05]),
 'mean_score_time': array([0.00180774, 0.00119705, 0.00159578, 0.00139689, 0.00160413,
        0.0016026 , 0.00138974, 0.00159621]),
 'std_score_time': array([0.00040555, 0.00039854, 0.00048856, 0.00048891, 0.00049673,
        0.00049316, 0.00049288, 0.00048886]),
 'param_criterion': masked_array(data=['gini', 'gini', 'gini', 'gini', 'entropy', 'entropy',
                    'entropy', 'entropy'],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_random_state': masked_array(data=[None, None, 1, 1, None, None, 1, 1],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_splitter': masked_array(data=['best', 'random', 'best', 'random', 'best', 'random',
                    'best', 'random'],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'criterion': 'gini', 'random_state': None, 'splitter': 'best'},
  {'criterion': 'gini', 'random_state': None, 'splitter': 'random'},
  {'criterion': 'gini', 'random_state': 1, 'splitter': 'best'},
  {'criterion': 'gini', 'random_state': 1, 'splitter': 'random'},
  {'criterion': 'entropy', 'random_state': None, 'splitter': 'best'},
  {'criterion': 'entropy', 'random_state': None, 'splitter': 'random'},
  {'criterion': 'entropy', 'random_state': 1, 'splitter': 'best'},
  {'criterion': 'entropy', 'random_state': 1, 'splitter': 'random'}],
 'split0_test_score': array([0.50160772, 0.51446945, 0.49839228, 0.50803859, 0.49517685,
        0.51446945, 0.49517685, 0.51768489]),
 'split1_test_score': array([0.46451613, 0.4516129 , 0.46451613, 0.46129032, 0.48064516,
        0.46451613, 0.47741935, 0.46451613]),
 'split2_test_score': array([0.48064516, 0.48709677, 0.48387097, 0.48709677, 0.48387097,
        0.46774194, 0.48387097, 0.48709677]),
 'split3_test_score': array([0.49354839, 0.50322581, 0.49354839, 0.48709677, 0.51290323,
        0.50645161, 0.51290323, 0.50322581]),
 'split4_test_score': array([0.47419355, 0.49032258, 0.48064516, 0.49677419, 0.48387097,
        0.49032258, 0.48709677, 0.47741935]),
 'mean_test_score': array([0.48290219, 0.4893455 , 0.48419459, 0.48805933, 0.49129343,
        0.48870034, 0.49129343, 0.48998859]),
 'std_test_score': array([0.01328322, 0.02123034, 0.01173894, 0.01545026, 0.01187969,
        0.02002956, 0.01222505, 0.01874835]),
 'rank_test_score': array([8, 4, 7, 6, 1, 5, 1, 3])}
```

Figura 24: resultado GridSearch

Naive Bayes

El segundo modelo que hemos implementado ha sido un modelo basado en el clasificador probabilístico de Naive Bayes, que se fundamenta en el teorema de Bayes con la asunción ingenua ("naive") de independencia condicional entre cada par de atributos dados la etiqueta de la clase.

Más en concreto vamos a hacer uso el clasificador de Naive Bayes gaussiano, *GaussianNB*. Hemos elegido este clasificador en concreto ya que nuestros datos estandarizados siguen una distribución Gaussiana o Normal.

Este clasificador tiene tan sólo dos parámetros como inputs los cuales son:

- *priors*: las probabilidades a priori de las clases.
- *var_smoothing*: porción de la variación más grande de todas las características, la cual se agrega a las varianzas para la estabilidad del

cálculo.

Al tener solamente dos inputs como parámetros, hemos considerado que no es necesario aplicar la Búsqueda por Rejilla en este clasificador ya que no es un factor que afecte significativamente al rendimiento del modelo.

3.4. EVALUANDO RESULTADOS

Estos resultados se han obtenido transformando los datos con el escalador *MinMaxScaler(0,1)*.

% (MinMaxScaler)	MODELOS	GridSearchCV	GridSearchCV Anidado	Sin GridSearch
CUOTAS_DB	DecisionTree	49.1	50.2	48.2
	NaiveBayes	-	-	54.1
CUOTAS_TEND5 (5 ult. part.)	DecisionTree	45.4	43.8	43.2
	NaiveBayes	-	-	54.4

En la columnas con GridSearch, es importante aclarar que se ha reflejado en la tabla el resultado del clasificador con los mejores hiperparámetros según el propio GridSearch.

En cúanto al árbol de decisión, viendo el árbol resultante[4], hemos deducido que está sobreentrenado, por lo que habrá que revisar los hiperparámetros óptimos para el modelo, como por ejemplo el número de elementos mínimo para formar una hoja.

En NaiveBayes hemos decidido considerado que no es necesario hacer una búsqueda con rejilla de sus hiperparámetros óptimos ya que considerando sus hiperparámetros, no creemos que tengan un impacto significativo en el resultado.

[4] deberia incluir el codigo para ver el arbol y el propio arbol???

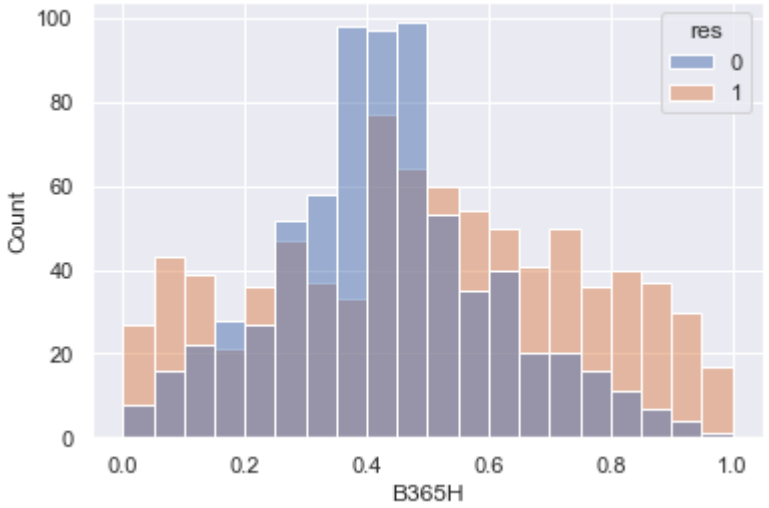

Figura 25: histograma que representa donde acierta y falla el NaiveBayes sin GridSearch en CuotasDB dependiendo de B365H (1 - acierto, 0 - fallo)

En este gráfico (Figura 25), se muestra la eficacia de NaiveBayes sin búsqueda por rejilla en nuestro primer dataset, CuotasDB. El resultado global era de 48.2% de acierto. Destacar como en los extremos el modelo rinde notablemente bien con una gran cantidad de muestras acertadas frente a las falladas. Estos extremos corresponden con partidos donde uno de los equipos es muy superior al otro, especialmente acertamos cuando el local es muy superior al visitante.

Sin embargo, en los valores centrales de la característica B365H, justo la zona donde se concentran la gran mayoria de muestras, el modelo tiene un porcentaje de fallo elevado, especialmente el rango de $0.3 < B365H < 0.5$.

Esto demuestra que simplemente con la información que nos proporcionan las cuotas no tenemos suficiente para predecir correctamente partidos más igualados. Para ello, en un futuro deberíamos incorporar nuevos datos que nos arrojen información independiente a la que tenemos sobre este tipo de choques.

La Figura 26, es otra representación, similar a la Figura 13, pero en este caso queremos ver la distribución de nuestros aciertos y fallos respecto a los dos atributos de cuotas que tenemos, B365H y B365A. La interpretación es la misma a la expuesta en la Figura 25.

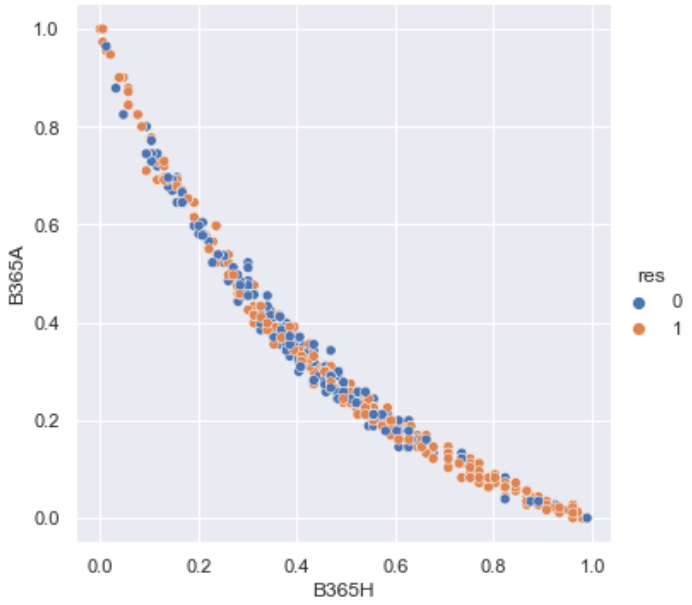

Figura 26: gráfica que representa el acierto y fallo del NaiveBayes sin GridSearch en CuotasDB dependiendo de B365H y **365A** (1 - acierto, 0 - fallo)

En el siguiente gráfico, Figura 27, vemos de nuevo un histograma que esta vez muestra la eficacia según el valor de la *dif_tend* de nuestro árbol de decisión entrenado sin GridSearch en nuestro dataset de partidos con información de las tendencias de los equipos en los últimos 5 partidos. La puntuación global bajaba en este caso a tan sólo un 43.2%.

Como vemos, en este caso el clasificador tiene una eficacia muy baja prácticamente en todos los valores de *dif_tend* tiene más fallos que aciertos. Esto puede ser debido a un sobreentrenamiento del árbol.

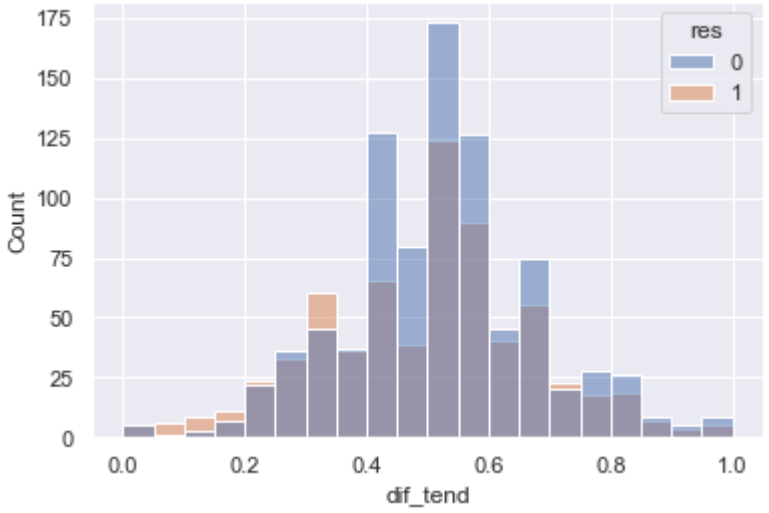

Figura 27: histograma que representa donde acierta y falla el DecissionTree sin GridSearch en CuotasDB_trend5 dependiendo de dif_tend (1 - acierto, 0 - fallo)

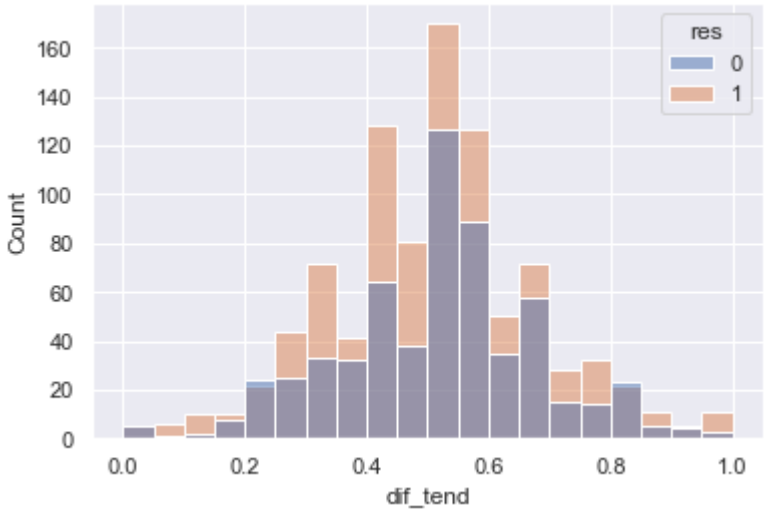

Figura 27: histograma que representa donde acierta y falla el NaiveBayes sin GridSearch en CuotasDB_trend5 dependiendo de dif_tend (1 - acierto, 0 - fallo)

4. CONCLUSIONES

Voy a listar y explicar brevemente varios aspectos que debería mejorar o desarrollar más en los próximos pasos:

- Manejar los *missing values*: los valores que faltan o incompletos debería descartarlos, por ejemplo, en el dataset de las tendencias, podriamos descartar los primeros N partidos ya que no tenemos registros.
- Probar otros escaladoree: realizar pruebas y validaciones con otros *scalers*, como *Normalizer*, *StandartScaler* o el *OneHotEncoder*.[5]
- Optimizar el Árbol de Decisión para que deje de estar sobreentrenado y se pueda utilizar en nuevos conjunto de datos. Para ellos debemos probar a entrenar otros hiperparámetros del árbol, como hemos nonmbrado anteriormente.
- Utilizar otras herramientas para visualizar los datos y los resultados. Especialmente hacer uso de la *Confusion matrix* para visualizar los resultados y entender en que aspecto y en que datos debe mejorar más nuestro modelo.
- Nuevos datasets. Además de manejar los *missing values*, es importante también probar nuevos datasets como por ejemplo añadir otro tipo de cuotas, probar con otra cantidad de partidos para la tendencia de un equipo, darle más peso a la tendencia de los partidos locales o visitantes, dependiendo de donde juegue el equipo, etc.

[5] más escaladores en https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing