Aprendizaje Automático Avanzado. Trabajo Final

Iraitz Azcarate

30 de diciembre de 2024

Índice

1. Introducción

El siguiente trabajo trata de estudiar el impacto que tienen las métricas de juego en el resultado final de partidos de fútbol en ligas profesionales.

Partiendo de diversos parámetros estadísticos recogidos durante el desarrollo de los partidos, trataremos de generar un modelo capaz de clasificar dichos partidos según su resultado.

A diferencia de los tan famosos modelos predictivos en el ámbito deportivo, en este caso el objetivo no será predecir, ya que necesitamos que el partido haya transcurrido para poder así acceder a los datos. Sin embargo, un modelo clasificador que determine si un partido finaliza con victoria local, victoria visitante o empate a partir de las métricas de juego puede ser de gran interés para obtener información sobre el impacto de estas métricas estadísticas en el resultado final del encuentro.

2. Datos

2.1. Descripción

```
path = kagglehub.dataset_download("bastekforever/complete-
    football-data-89000-matches-18-leagues")
filename = os.listdir(path)[0]
file_path = os.path.join(path, filename).replace('\\','/')
df = pd.read_csv(file_path)
```

Para llevar a cabo el análisis propuesto vamos a hacer uso de la base de datos llamada *Football DataSet +96k matches (18 leagues)*. Este dataset proviene de *Kaggle*, y nos aporta diversos datos sobre partidos de futbol profesional. Recoge partidos de 18 ligas europeas, entre las que se encuentran las 5 grandes ligas y sus respectivas segundas divisiones. Además, contiene un gran volumen de datos, ya que aporta partidos desde principios de los años 2000 hasta la actualidad.

Liga	Temporada	Partidos
Premier League	2002-2022	7600
La Liga	2003-2022	7220
Serie A	2003-2022	7150
Ligue 1	2004-2022	6757
Championship	2010-2022	6684
League One	2010-2022	6440
Bundesliga	2003-2022	5838
League Two	2011-2022	6015
Eredivisie	2004-2022	5776
La Liga 2	2010-2022	5519
Serie B	2010-2022	5286
Ligue 2	2010-2022	4470
Super Lig	2010-2022	3504
Jupiler League	2010-2022	3756
Fortuna 1 Liga	2010-2022	3687
2. Bundesliga	2010-2022	3503
Liga Portugal	2010-2022	3414
PKO BP Ekstraklasa	2010-2022	3338

Cuadro 1: Tabla de ligas y partidos por temporada

Con respecto a las variables recogidas, disponemos de un total de 56 parámetros:

Variable	Descripción
League	Liga a la que pertenece el partido
Home	Nombre del equipo local
Away	Nombre del equipo visitante
INC	Incidentes registrados en el partido
Round	Jornada o ronda del campeonato
Date	Fecha del partido
Time	Hora del inicio del partido
H_Score	Goles marcados por el equipo local
A_Score	Goles marcados por el equipo visitante
HT_H_Score	Goles del equipo local al descanso
HT_A_Score	Goles del equipo visitante al descanso
WIN	Equipo ganador del partido
H_BET	Probabilidad de victoria del equipo local según apuestas
X_BET	Probabilidad de empate según apuestas
A_BET	Probabilidad de victoria del equipo visitante según apuestas
WIN_BET	Resultado ganador basado en apuestas
OVER_2.5	Si el total de goles supera los 2.5
OVER_3.5	Si el total de goles supera los 3.5

Cuadro 2: Descripción de Variables (Parte 1)

Variable	Descripción
H_15	Goles del equipo local en los primeros 15 minutos
A_15	Goles del equipo visitante en los primeros 15 minutos
H_45_50	Goles del equipo local entre los minutos 45 y 50
A_45_50	Goles del equipo visitante entre los minutos 45 y 50
H_90	Goles totales del equipo local
A_90	Goles totales del equipo visitante
H_Missing_Players	Jugadores ausentes del equipo local
A_Missing_Players	Jugadores ausentes del equipo visitante
Missing_Players	Total de jugadores ausentes en el partido
H_Ball_Possession	Posesión del balón del equipo local
A_Ball_Possession	Posesión del balón del equipo visitante
H_Goal_Attempts	Intentos de gol del equipo local
A_Goal_Attempts	Intentos de gol del equipo visitante
H_Shots_on_Goal	Tiros a puerta del equipo local
A_Shots_on_Goal	Tiros a puerta del equipo visitante
H_Attacks	Ataques totales del equipo local
A_Attacks	Ataques totales del equipo visitante
H_Dangerous_Attacks	Ataques peligrosos del equipo local
A_Dangerous_Attacks	Ataques peligrosos del equipo visitante
H_Shots_off_Goal	Tiros fuera del equipo local
A_Shots_off_Goal	Tiros fuera del equipo visitante
H_Blocked_Shots	Tiros bloqueados al equipo local
A_Blocked_Shots	Tiros bloqueados al equipo visitante
H_Free_Kicks	Tiros libres a favor del equipo local
A_Free_Kicks	Tiros libres a favor del equipo visitante
H_Corner_Kicks	Saques de esquina del equipo local
A_Corner_Kicks	Saques de esquina del equipo visitante
H_Offsides	Fuera de juego del equipo local
A_Offsides	Fuera de juego del equipo visitante
H_Throw_in	Saques de banda del equipo local
A_Throw_in	Saques de banda del equipo visitante
H_Goalkeeper_Saves	Paradas del portero del equipo local
A_Goalkeeper_Saves	Paradas del portero del equipo visitante
H_Fouls	Faltas cometidas por el equipo local
A_Fouls	Faltas cometidas por el equipo visitante
H_Yellow_Cards	Tarjetas amarillas para el equipo local
A_Yellow_Cards	Tarjetas amarillas para el equipo visitante
Game Link	Enlace al partido

Cuadro 3: Descripción de Variables (Parte 2)

Como vemos, la base de datos nos aporta tanto datos estadísticos del partido como otro tipo de datos entre los que encontramos información sobre casas de apuestas e información general sobre el partido. Para nuestro estudio, queremos

partir de datos totalmente estadísticos, basados en las métricas de juego, por lo que eliminaremos todas las columnas que consideramos no útiles para el objetivo final.

Además, eliminamos los parámetros relativos al resultado y nos quedamos con un único parámetro *Result* que indicará el resultado final del encuentro. De esta forma, intentamos atribuir un resultado a cada partido partiendo únicamente de las métricas de juego, sin tener información previa sobre el resultado en ningún momento del partido.

Por razones similares, eliminamos los datos respectivos a los nombres de los equipos, ligas y rondas, ya que no queremos que estos datos influyan en nuestros modelos.

```
columnas_a_eliminar =  ["Home", "Away", "INC", "Date", "
    Round", "League", "Time", "HT_H_Score", "HT_A_Score", "
    WIN", "H_BET", "X_BET", "A_BET", "WIN_BET", "H_15", "A_15
    ", "H_45_50", "A_45_50", "H_90", "A_90", "Game Link"]
df = df.drop(columns=columnas_a_eliminar)
```

Con esto, reducimos la dimensión de nuestros datos, pasando de tener 56 parámetros a 35.

Una vez eliminadas las columnas consideradas poco apropiadas para nuestro análisis, debemos de identificar los parámetros que usaremos como etiqueta para entrenar los modelos de clasificación.

En este caso, estamos interesados en clasificar cada partido por su resultado final.

Por lo tanto, distinguimos 3 categorías:

- Victoria Local (1)
- Empate(X)
- Victoria Visitante (2)

Sin embargo, en nuestro conjunto de datos no tenemos esta información específicamente, por lo que añadiremos un nuevo parámetro *Result* basándonos en

la información de los parámetros *H_Score* y *A_Score*.

```
df['Result'] = np.where(df["H_Score"] > df["A_Score"], '1',
                np.where(df["H_Score"] < df["A_Score"], '2',
                   "x"))
df = df.drop(columns=['H_Score', 'A_Score'])
```

De esta manera, nuestro dataset inicial contiene 96.337 instancias, 33 variables predíctoras y 1 variable categórica a la que llamaremos variable etiqueta.

2.2. Limpieza

Una vez tenemos nuestro dataset bien estructurado con los parámetros predictores y la variable etiqueta bien ubicados, es hora de dar un paso hacia dentro de los datos.

Lo primero es ver cual es la cantidad real de nuestros datos. A pesar de conocer el número de parámetros e instancias de nuestro dataset, este puede estar parcialmente vació debido a la presencia de valores nulos. En estos casos, debemos identificar las instancias o variables afectadas y decidir como actuar con el objetivo de minimizar el impacto de estos datos vacíos en nuestro análisis.

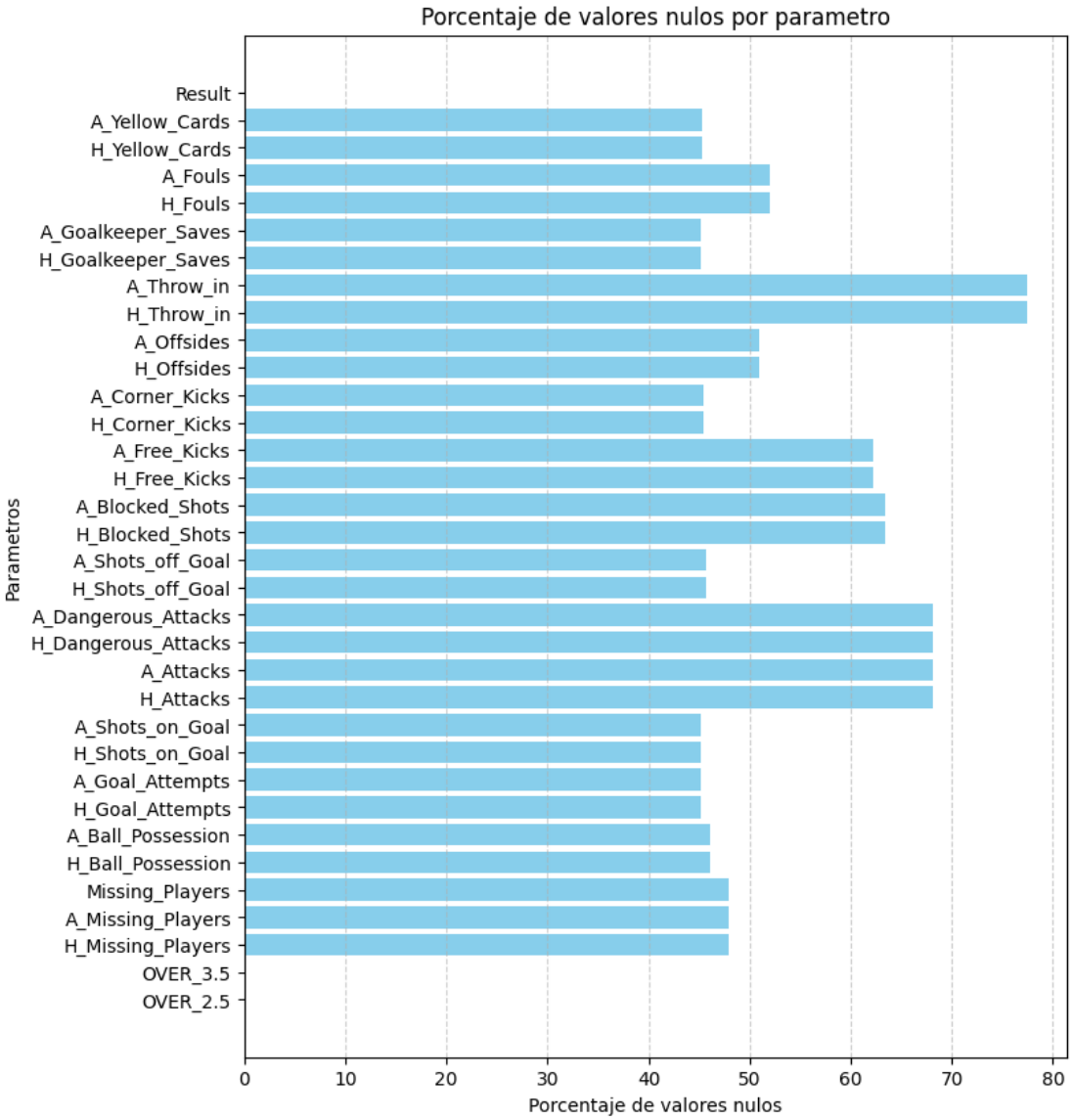

Figura 1: Porcentaje de valores nulos de los datos.

Como vemos en el gráfico, tenemos un porcentaje alto de valores nulos en todos los parámetros predictores relacionados con métricas estadísticas del juego, con la excepción de las métricas *OVER_2.5* y *OVER_3.5* que provienen de la rama de métricas relacionadas con apuestas. Por suerte, la variable *Result* no contienen valores nulos, y por lo tanto, disponemos de todas las etiquetas.

Para empezar, eliminaremos los partidos que tengan más de un 30 % de valores nulos. Con suerte, conseguiremos que el número de valores nulos disminuya considerablemente.

```
umbral = int(df.shape[1] * 0.7)
df_filtered_1 = df.dropna(thresh=umbral)
```

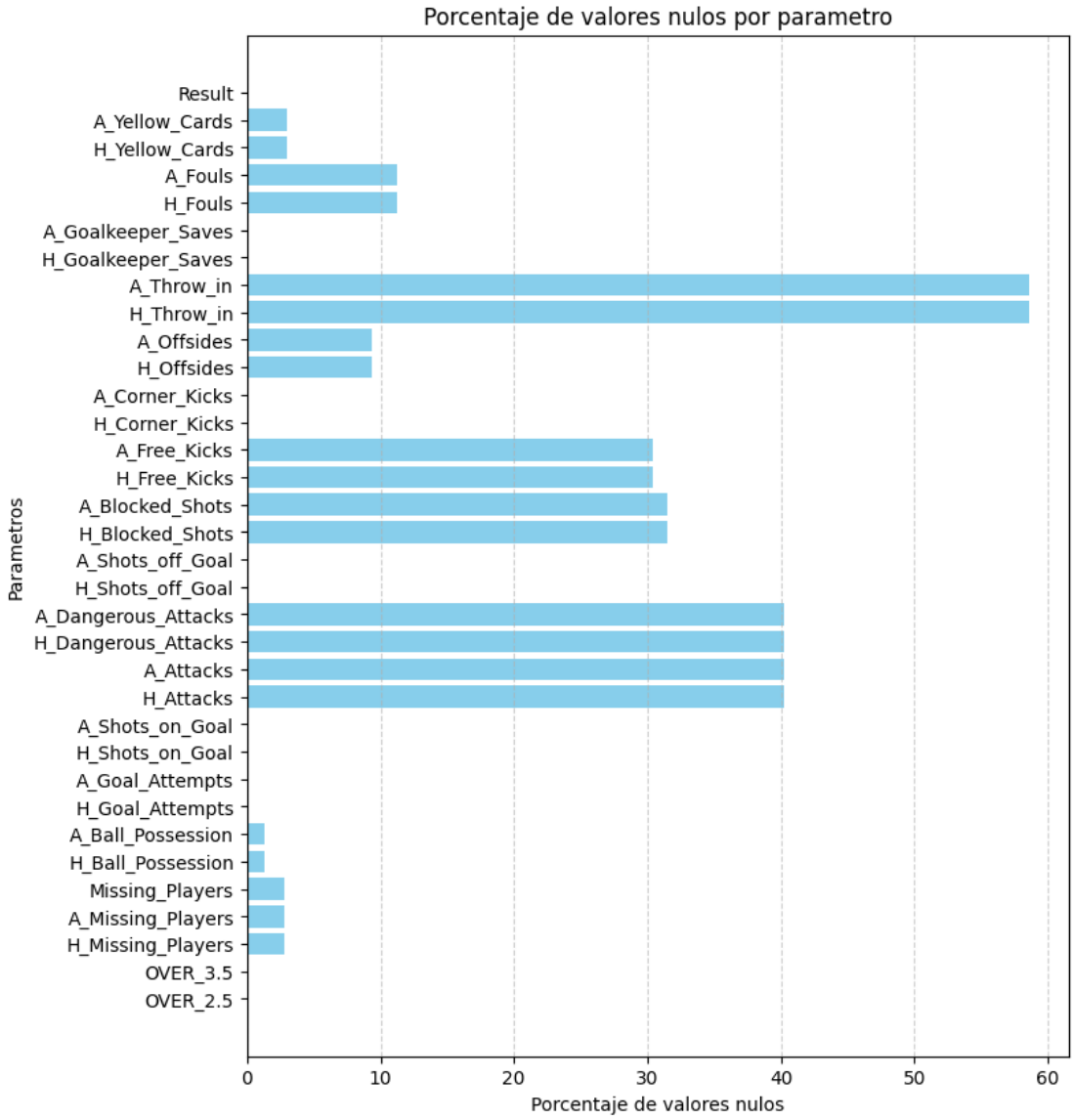

Figura 2: Porcentaje de valores nulos de los datos tras primera limpieza.

Tras esta primera criba, pasamos de tener 96.337 datos a poco más de la mitad (51.347). Aun y todo, seguimos teniendo bastantes datos vacíos en nuestro dataset.

Destacan notablemente 10 columnas en las que el número de valores nulos supera el 30 % del total. En este caso, nos decantamos por eliminar estas columnas.

```
columnas_a_eliminar = ['A_Throw_in', 'H_Throw_in', '
    A_Free_Kicks', 'H_Free_Kicks', 'A_Blocked_Shots', '
    H_Blocked_Shots', 'A_Dangerous_Attacks', '
    H_Dangerous_Attacks', 'A_Attacks', 'H_Attacks']
df_filtered_2 = df_filtered_1.drop(columns=
    columnas_a_eliminar)
```

Finalmente, para el resto de variables, reemplazamos los valores nulos por los valores medios de cada parámetro. En este caso, utilizaremos las medias estratificadas de cada variable, calculadas por clase de nuestra variable etiqueta.

```
for column in df_filtered_2.select_dtypes(include=[float,
    int]).columns:
     df_filtered_2[column] = df_filtered_2.groupby('Result')[
        column].transform(lambda x: x.fillna(x.mean()))
```

Con todo esto, logramos tener liberarnos de absolutamente todos los datos nulos de nuestro dataset. El volumen de datos final es de 23 variables predictoras, 1 variable etiqueta y 51.347 datos.

2.3. Transformación

Una vez tenemos los datos totalmente completos, debemos de valorar la estandarización. En el caso de que los parámetros predictores sean de escala diferente, es muy posible que los de mayor escala tomen una mayor importancia en nuestro modelo, mientras que aquellos predictores con menor escala no tendrán apenas impacto en el análisis. Para evitarlo, a cada variable, se le resta su media y divide por la desviación estándar, obteniendo variables con media 0 y desviación 1.

Veamos si en nuestro caso es necesario.

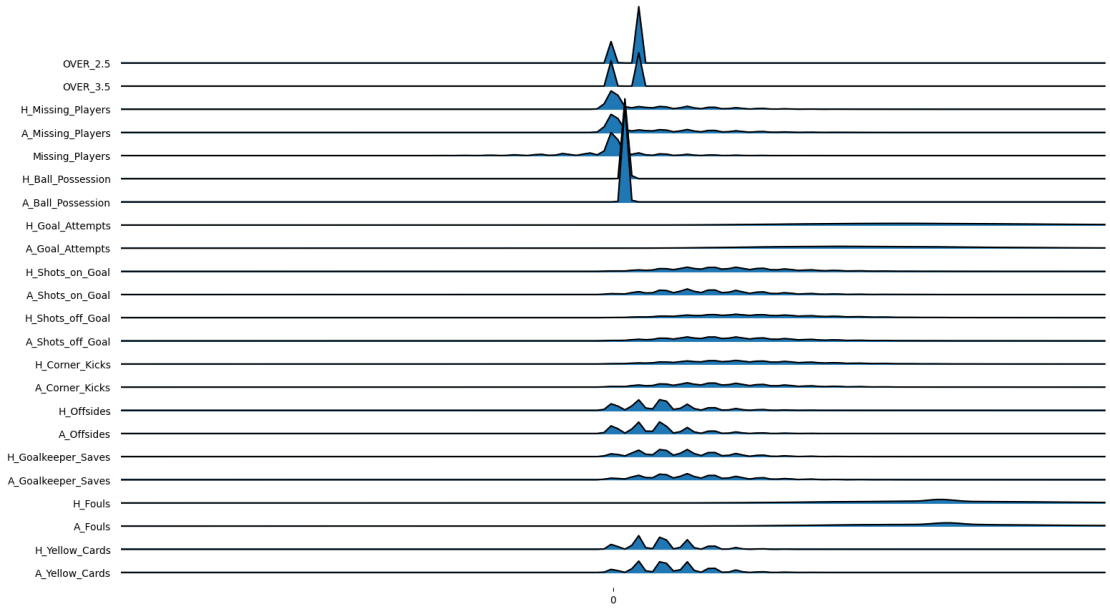

Figura 3: Distribuciones de las variables predictoras.

Como vemos, hay mucha diferencia entre las distribuciones de los distintos parámetros predictores, por lo que nos decantamos por normalizar las variables de nuestro dataset.

Sin embargo, también podemos apreciar en el gráfico anterior que muchas de ellas tienen distribuciones con varios máximos y mínimos locales. Esto se debe posiblemente a que se trata de variables enteras, discretas. Por ello, antes de estandarizar los datos, vamos a realizar una transformación *Box-Cox* con el objetivo de aproximar estas distribuciones a distribuciones normales, minimizando la perdida de varianza.

Como el algoritmo original de *Box-Cox* no es aplicable a datos no estrictamente positivos, aplicaremos la transformación de *Yeo Jonhson*. Esta transformación es una generalización de la anterior para valores reales, y sigue la siguiente definición:

Para valores $y \geq 0$:

$$T(y;\lambda) = \begin{cases} \frac{(y+1)^{\lambda}-1}{\lambda}, & \text{si } \lambda \neq 0, \\ \ln(y+1), & \text{si } \lambda = 0. \end{cases}$$

Para valores $y < 0$:

$$T(y;\lambda) = \begin{cases} \frac{-((-y+1)^{2-\lambda}-1)}{2-\lambda}, & \text{si } \lambda \neq 2, \\ -\ln(-y+1), & \text{si } \lambda = 2. \end{cases}$$

Este parametro λ sera ajustado automaticamente por la funcion implementada.

```
from sklearn.preprocessing import PowerTransformer

# df_numeric contiene los datos numericos de mi dataset
df_numeric = df_filtered_2.select_dtypes(include=['float64',
    'int64'])
df_no_numeric = df_filtered_2.drop(columns = df_numeric.
   columns)
# Crear el objeto PowerTransformer con Yeo-Johnson
pt = PowerTransformer(method='yeo-johnson')

# Ajustar y transformar los datos
df_transformed = pt.fit_transform(df_numeric)

# Convertir el resultado en un DataFrame
df_transformed = pd.DataFrame(df_transformed, columns=
   df_numeric.columns)

# Ahora 'df_transformed' contiene los datos transformados
df_transformed = df_transformed.reset_index(drop=True)
df_no_numeric = df_no_numeric.reset_index(drop=True)
df_filtered_3 = pd.concat([df_transformed, df_no_numeric],
   axis=1)
```

Finalmente, normalizamos las variables predictoras

```
numeric_columns = df_filtered_3.select_dtypes(include=[float
   , int]).columns
df_filtered_3[numeric_columns] = (df_filtered_3[
   numeric_columns] - df_filtered_3[numeric_columns].mean())
    / df_filtered_3[numeric_columns].std()
```

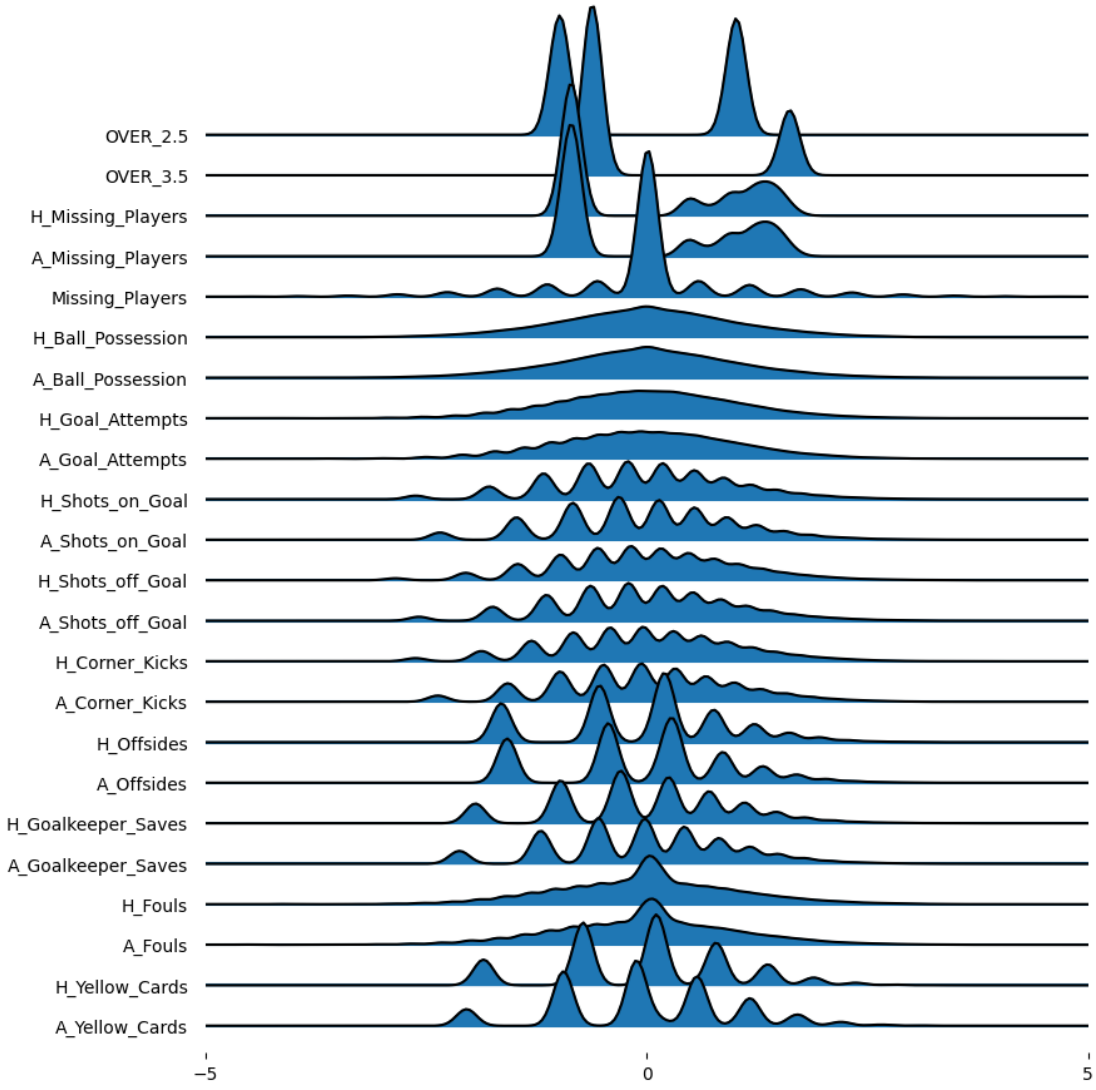

Figura 4: Distribuciones de las variables predictoras tras aplicar transformaciones.

2.4. Balanceo de clases

El siguiente paso en nuestro análisis será ver si las clases de nuestro parámetro etiqueta están bien balanceadas, es decir, si el número de instancias no varía en exceso entre las diferentes clases. En nuestro caso, al tratarse de resultados de partidos de futbol, podemos predecir un pequeño desbalanceo a favor de las victorias de equipos locales, pero no debería ser desproporcionado. Comprobémoslo.

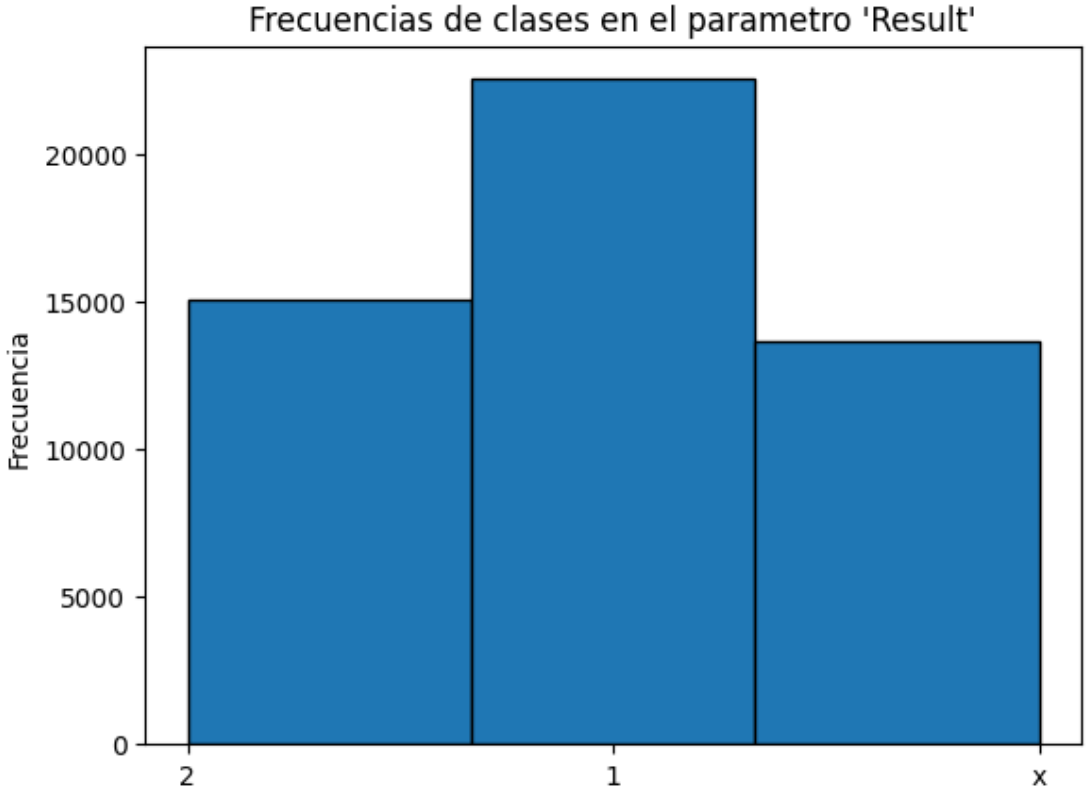

Figura 5: Distribución de los datos según clases de la variable etiqueta.

Como vemos en el histograma, y tal y como habíamos predicho, existe una pequeña descompensación en el número de instancias según el resultado del partido. Sin embargo, esta diferencia no es ni mucho menos excesiva, y por lo tanto, no es necesario aplicar técnicas de resampleo para generar datos de las clases minoritarias.

2.5. Feature extraction

A continuación, vamos a analizar la información que nos aportan las diferentes variables predictoras de nuestro dataset. Como es bien sabido, una alta dimensionalidad de los datos puede suponer un problema a la hora de obtener resultados, sobre todo al trabajar con algoritmos de Machine Learning clásicos. Por ello, vamos a estudiar cuanta información (varianza) nos proporcionan las distintas variables, y así poder eliminar parámetros poco útiles si los hubiera.

Empecemos viendo la matriz de correlaciones de las variables predictoras.

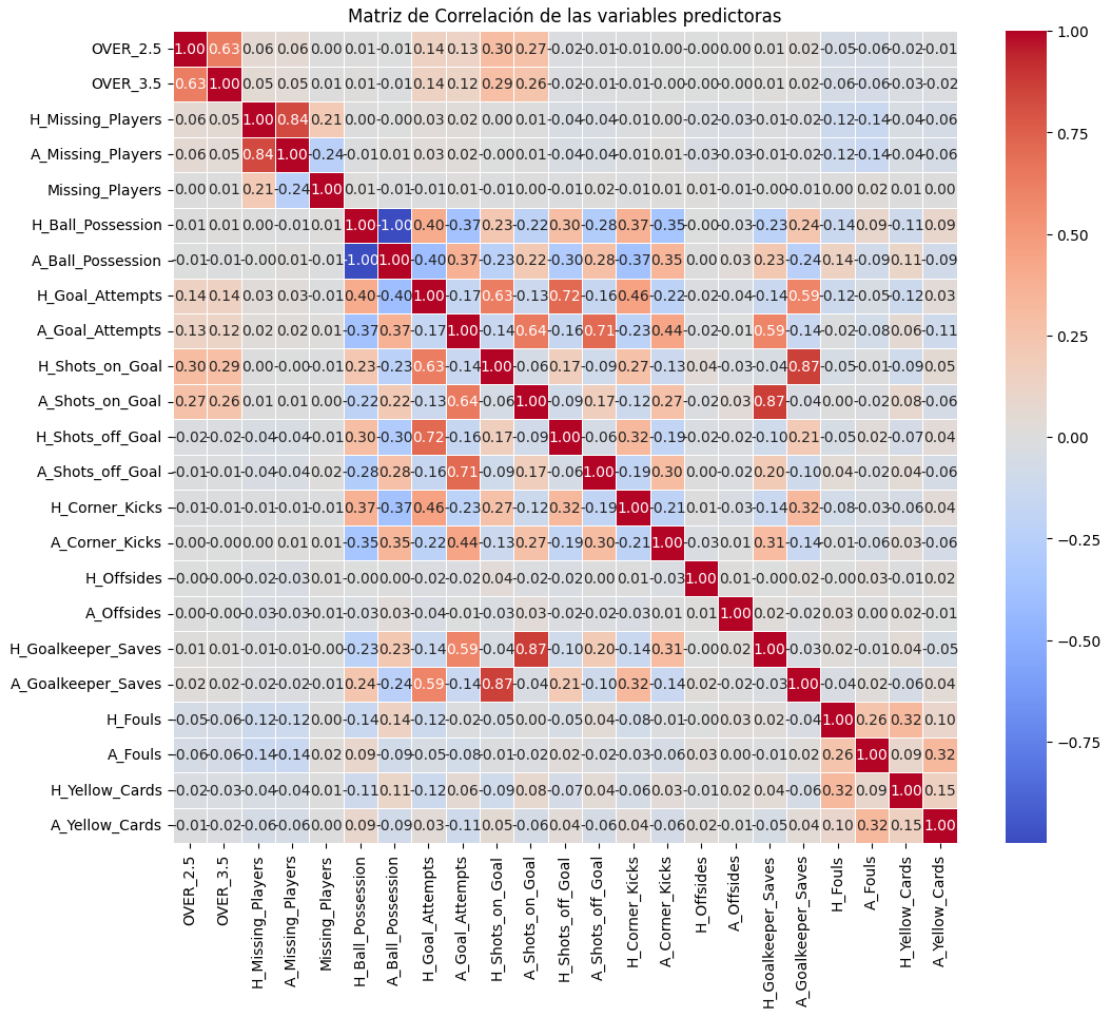

Figura 6: Matriz de correlación de las variables predictoras.

Como podemos observar, hay correlaciones de todo tipo entre las variables predictoras.

Lo primero que nos llama la atención son los valores de *Corr(H_Ball_Possession, A_Ball_Possession) = -1*. Como es lógico, son valores cuya suma debe de ser igual a 100 (antes de normalizar), por lo que el valor de uno de los parámetros determina unívocamente el del segundo. Por ello, eliminamos uno de los valores, en este caso *A_Ball_Possession*.

Por otro lado, tenemos dos pares de correlaciones muy altas *Corr(H_Goalkeeper_Saves, A_shots_on_Goal) = Corr(A_Goalkeeper_Saves, H_shots_on_Goal) = 0.88*. En este caso, es claro que el número de paradas de un portero esta directamente relacionado con el número de disparos a puerta del equipo contrario. Es más, en este caso, cabe la posibilidad de que los modelos clasificadores tomen estas dos variables

para calcular el número de goles que ha anotado cada equipo y así determinar el resultado del partido. Como esto no es algo que nos interese en este estudio, ya que restaria importancia a todas las demás métricas de juego, optamos por eliminar las variables *H_Goalkeeper_Saves* y *A_Goalkeeper_Saves*.

```
df_filtered_3 = df_filtered_3.drop(columns = ['
    A_Ball_Possession', 'H_Goalkeeper_Saves', '
    A_Goalkeeper_Saves'])
```

Con lo que respecta a las demás correlaciones, considero que no son lo suficientemente altas (en valor absoluto) para considerar el eliminar más variables.

Tras eliminar estas variables altamente correlacionadas con otras variables predictoras, vamos a ver si podemos expresar la información de nuestros datos con un menor número de variables.

Para ello, vamos a aplicar la técnica del análisis de componentes principales (PCA). Esta se basa en proyectar los datos multidimensionales sobra la dirección que maximiza su varianza. Esta proyección de los datos será la primera componente principal. A continuación, y de manera repetida, se vuelven a proyectar los datos sobre una dirección ortogonal a las anteriores componentes principales, nuevamente maximizando la varianza de los datos. De esta forma, logramos capturar la mayor información (variabilidad) de nuestros datos en el menor número de dimensiones posibles.

Apliquemos esta técnica a nuestros datos.

```
df_pred = df_filtered_3.drop(columns=['Result'])
pca = PCA()
pca_result = pca.fit_transform(df_pred)

# Varianza explicada por cada componente
explained_variance_ratio = pca.explained_variance_ratio_
explained_variance_cumulative = np.cumsum(
    explained_variance_ratio)
```

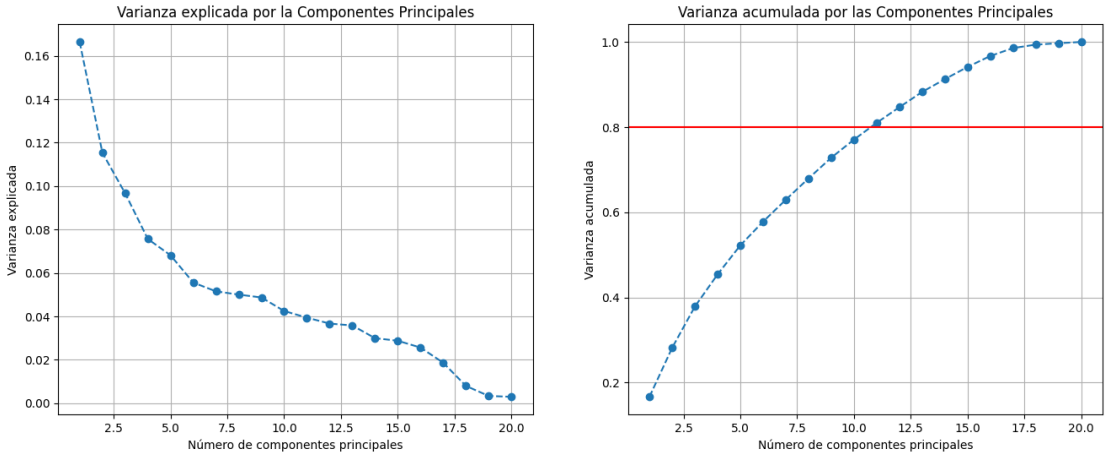

Figura 7: Resultados del Analisis de Componentes Principales.

Como se puede observar en los gráficos mostrados, no es posible reducir significativamente la dimensionalidad de nuestros datos sin perder información. Para preservar un 80 % de varianza, debemos trabajar con las primeras 11 componentes principales. Ademas, el transformar nuestros datos de esta manera dificulta la posterior interpretabilidad de los modelos, lo cual es vital para los objetivos de este análisis. Por ello, partiremos con las variables predictoras en crudo, y de no tener éxito, probaremos posteriormente con dichas componentes principales.

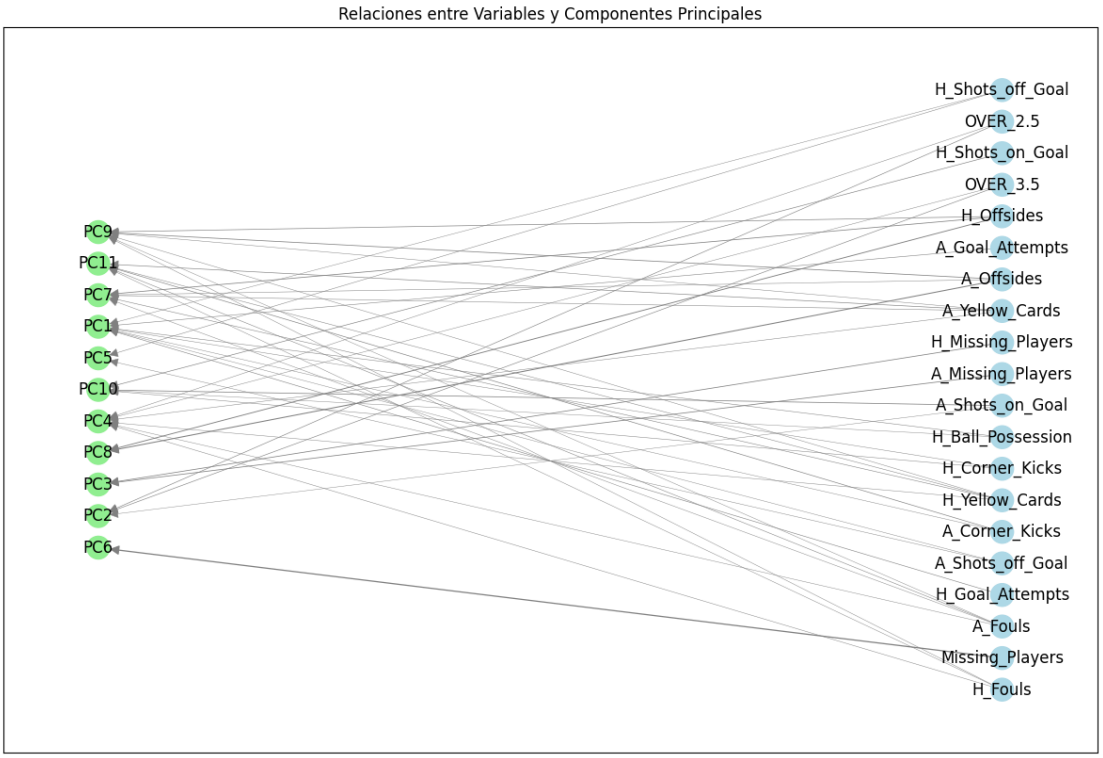

Figura 8: Principales relaciones entre las variables predictoras y las primeras componentes principales.

Como podemos observar en el gráfico anterior, la interpretabilidad de las componentes puede suponer un problema en este caso.

Por ejemplo, podemos ver que las variables predictoras con más peso en la primera componente principal son: *A_Corner_kicks*, *H_Corner_kicks*, *H_Ball_Possession*, *H_Goal_Attemps*, *A_Goal_Attemps* y *H_Shots_off_Goal*, lo cual no nos da una idea general de lo que interpreta dicha componente.

2.6. Detección de Outliers

Otro de las cuestiones a trabajar antes de aplicar cualquier tipo de modelo es la detección de outliers. Los outliers son datos que están alejados del resto, fuera de lo que llamamos *Nube de puntos* formada por las instancias del dataset. Hay muchas formas de definir lo que es 'estar lejos del resto', ya que depende de la distancia empleada y de como se aplica dicha distancia entre un punto y un conjunto de puntos.

En este caso, comenzaremos por un algoritmo sencillo, el llamado *Local Outlier Factor (LOF)*. Este considera las distancias euclideas entre cada punto y sus k vecinos más próximos, y en función a estas distancias asigna un valor de *outlierness*. De esta forma, las instancias con valor de outlierness mayor a un umbral previamente fijado serán consideradas outliers.

```
np.random.seed(42)
clf = LocalOutlierFactor(n_neighbors=5)

clf.fit_predict(df_pred)
outlierness = clf.negative_outlier_factor_
df_pred["outlierness"] = abs(outlierness)

sns.kdeplot(df_pred["outlierness"], color="blue", fill=False
    )
```

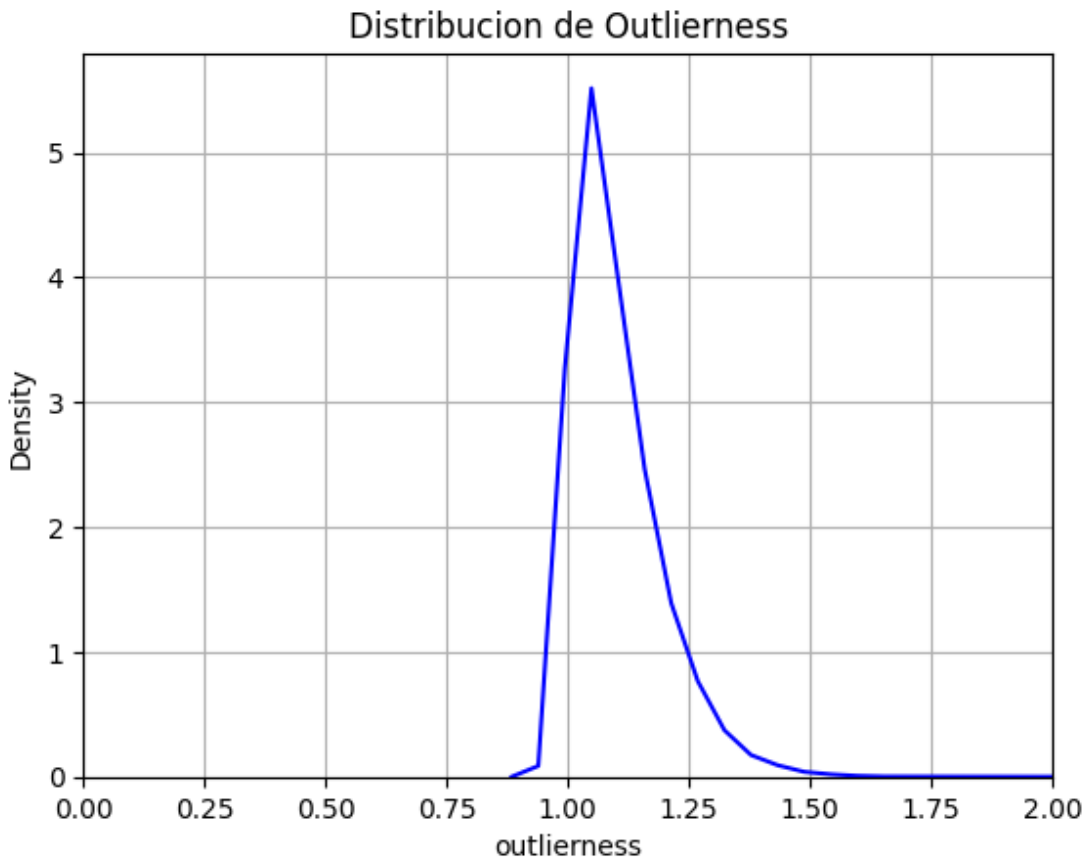

Figura 9: Distribución del valor de Outlierness de los datos.

En este caso, no parece haber excesivos puntos outliers. Podemos ver que de más de 50.000 datos, solo 3 superan el valor de outlierness = 2. En este caso, dado que tenemos una gran cantidad de datos, probamos con un valor de outlierness de 1.5, y de esta forma, obtenemos 118 puntos outlier.

Veamos a continuación que puntos hemos detectado como outliers. En este caso, dada la alta dimesionalidad de los datos y lo dividida que esta la varianza entre las variables predictoras, no será tarea simple visualizar estos outliers y ver el porque se han detectado. Probemos con la técnica de *t-distributed Stochastic Neighbor Embedding (TSNE).*

```
from sklearn.manifold import TSNE

df_pred = df_pred.reset_index(drop=True)
df_filtered_3 = df_filtered_3.reset_index(drop=True)
df_pred_complete = pd.concat([df_pred, df_filtered_3['Result
    ']], axis=1)

#Vamos a hacer uso de una muestra reducida de los datos para
    este proceso, ya que debido al alto volumen, el
    algoritmo TSNE es lento y los resultados no son muy
    visuales.
outliers = df_pred_complete[df_pred_complete['outlierness']
    > 1.5]
not_outliers = df_pred_complete[df_pred_complete['
    outlierness'] <= 1.5]
sample_df = not_outliers.sample(n=1000, replace=True,
    random_state=42)
sample_df = pd.concat([sample_df, outliers], ignore_index=
    True)
outliers = sample_df['outlierness'] > 1.5
result = sample_df['Result']
sample_df=sample_df.drop(columns=['outlierness','Result'])

# Configurar t-SNE
tsne = TSNE(n_components=2, perplexity=30, learning_rate
    =200, random_state=42)
# Reducir la dimensionalidad
data_tsne = tsne.fit_transform(sample_df)
```

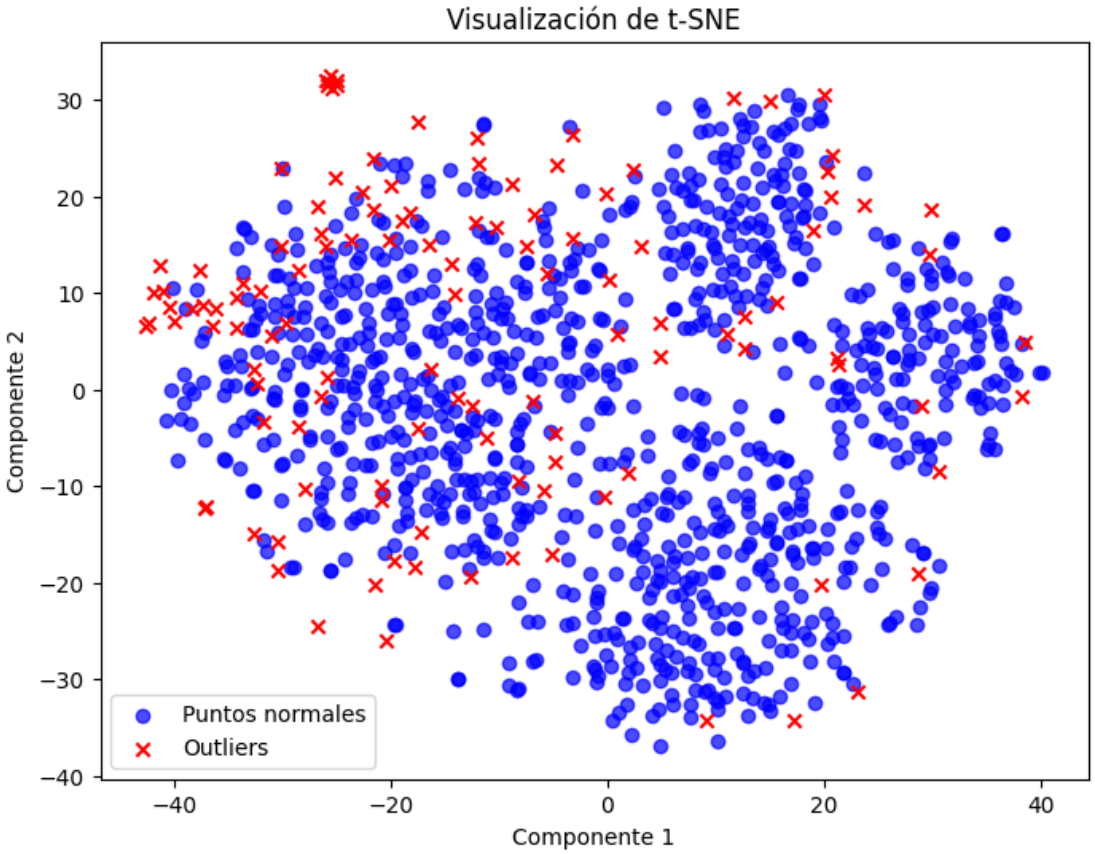

Figura 10: Visualización de Outliers mediante TSNE.

Como predecíamos, el TSNE no es capaz de mostrar la mayor parte de la información que contienen nuestro datos, y por lo tanto, no podemos ver realmente si los puntos marcados como outliers están realmente distanciados o no del resto.

Sin embargo, a pesar de todo si que podemos intuir como hay una mayor densidad de puntos outliers en la zona exterior de la denominada nube de puntos, especialmente en los valores más negativos de la primera componente. Aun así, por la naturaleza de nuestros datos, es muy complicado asegurar que realmente los outliers son puntos que realmente queremos eliminar. Aun y todo, decidimos eliminarlos debido al alto volumen de datos que poseemos y al pequeño número de outliers.

```
1 df_filtered_3 = df_filtered_3[df_pred["outlierness"]<1.5]
```

```
# Crear un mapa de colores para los valores unicos de 'Result'
unique_results = result.unique()
color_map = {result: plt.cm.tab10(i / len(unique_results)) for i, result in enumerate(unique_results)}

sample_df = sample_df.reset_index(drop=True)
result = result.reset_index(drop=True)
sample_df = pd.concat([sample_df, result], axis=1)
```

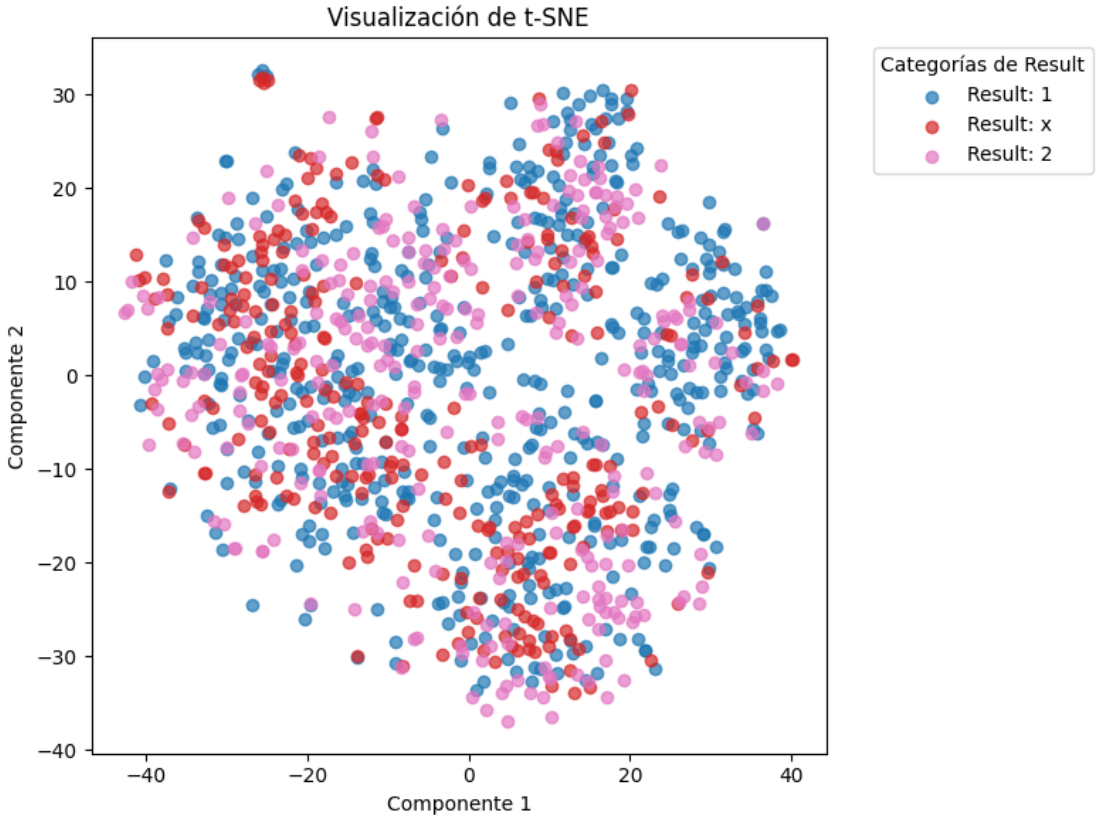

Figura 11: Visualización de las distintas clases de la variable etiqueta mediante TSNE.

Haciendo uso de la proyección realizada mediante el TSNE, visualizamos los datos según su valor en la variable etiqueta. Como vemos, esta representación no nos ayuda para distinguir grupos, seguramente debido al bajo porcentaje de variabilidad que recoge.

2.7. Selección de variables

Finalmente, como último paso previo a la aplicación de modelos de clasificación sobre nuestros datos, vamos a aplicar alguna técnica de selección de variables con el fin de reducir, si es conveniente, la dimensión de los datos.

Como primera tentativa, no podemos aplicar un *filter correlation by t-test*, ya que este solo es apto para problemas de clasificación binaria. En nuestro caso, probamos con un análisis de varianza (*ANOVA test*). Sin embargo, este test requiere normalidad en nuestros datos, no solo para cada variable predictora, sino para cada variable dentro de cada grupo *1*, *x* y *2*. Veamos si nuestros datos cumplen estas hipótesis.

Para ello, aplicamos un test de normalidad por cada grupo dentro de cada variable predictora. Por ejemplo, haciendo uso del test de *Shapiro-Wilk*:

- H_0: Los datos siguen una distribución normal.
- H_1: Los datos no siguen una distribución normal.

```
from scipy.stats import shapiro

X = df_filtered_3.drop(columns="Result")
y = df_filtered_3["Result"]

for column in X.columns:
    print(f"Feature: {column}")
    for label in ['1', 'x', '2']:
        values = X.loc[y == label, column]

        # Prueba de Shapiro-Wilk
        stat, p_value = shapiro(values)
        print(f"  Class {label} - p-value: {p_value}")
```

En nuestro caso, no podemos asumir normalidad en nuestros datos, por lo que no podemos aplicar el test de ANOVA. En su lugar, aplicamos el test no paramétrico de *Kruskal-Wallis*.

Este test comprueba si cada variable predictora difiere en su distribución para cada grupo de la variable etiqueta. En el caso de no diferir, esta variable será poco útil a la hora de aplicar un modelo clasificador. Por el contrario, si la distribución de esta variable cambia significativamente según cambia el valor de la variable etiqueta, esta variable será de gran valor a la hora de establecer el modelo final.

- H_0: Distribución_1 = Distribución_x = Distribución_2
- H_1: $\exists\, i, j \in \{1, x, 2\}$ tal que Distribución_$i \neq$ Distribución_j

```
from scipy.stats import kruskal

significant_features = []
p_value_threshold = 0.05

for column in X.columns:
    # Separar valores para cada clase
    class_1_values = X.loc[y == '1', column]
    class_x_values = X.loc[y == 'x', column]
    class_2_values = X.loc[y == '2', column]

    # Realizar test kruskal
    t_stat, p_value = kruskal(class_1_values, class_x_values
       , class_2_values)
    print(f"Feature: {column}, p-value: {p_value}")

    # Seleccionar si p-valor es menor al umbral
    if p_value < p_value_threshold:
        significant_features.append(column)

Resultados:
Feature: OVER_2.5, p-value: 0.0
Feature: OVER_3.5, p-value: 5.49448616357553e-39
Feature: H_Missing_Players, p-value: 5.438364178862826e-14
Feature: A_Missing_Players, p-value: 9.492421823890057e-16
Feature: Missing_Players, p-value: 0.33751146873017573
Feature: H_Ball_Possession, p-value: 0.00060892570063656668
Feature: H_Goal_Attempts, p-value: 2.945063319309058e-240
Feature: A_Goal_Attempts, p-value: 0.0
```

```
Feature: H_Shots_on_Goal, p-value: 0.0
Feature: A_Shots_on_Goal, p-value: 0.0
Feature: H_Shots_off_Goal, p-value: 6.8038662580837454e-18
Feature: A_Shots_off_Goal, p-value: 0.0021208628577805504
Feature: H_Corner_Kicks, p-value: 4.511451673682282e-16
Feature: A_Corner_Kicks, p-value: 0.8286732538472474
Feature: H_Offsides, p-value: 2.3541248197219893e-86
Feature: A_Offsides, p-value: 1.2096426248466608e-19
Feature: H_Fouls, p-value: 2.8288719434264626e-16
Feature: A_Fouls, p-value: 1.174960875951077e-31
Feature: H_Yellow_Cards, p-value: 9.292849797856982e-137
Feature: A_Yellow_Cards, p-value: 1.7785302340186362e-26
```

Por lo obtenido en los resultados de este análisis, existen dos variables en nuestro dataset que no tienen correlación significativa con nuestra variable etiqueta: *Missing_Players* y *A_Corner_Kicks*. Sin embargo, nos resulta extraño que la variable *A_Corner_Kicks* tenga un p-valor tan alto cuando su variable hermana *H_Corner_Kicks* tiene un p-valor muy cercano al 0. Por ello, vamos a aplicar una segunda técnica para comprobar estos resultados.

En este caso, vamos a aplicar el enfoque de *Correlation Feature Selection*. El objetivo es seleccionar las variables predictoras más relevantes dentro de nuestro conjunto de datos, evitando una alta correlación entre estas variables para no tener información redundante. Para ello se hace uso del heurístico de merito *Merit*

$$Merit(S) = \frac{k \cdot \overline{r_c}}{\sqrt{k + k(k-1)\overline{r_r}}}$$

Donde:

- k es el número de características seleccionadas en el subconjunto S.
- $\overline{r_c}$ es el promedio de la correlación entre las características de S y la variable objetivo.
- $\overline{r_r}$ es el promedio de las correlaciones entre las características dentro de S.

En esta fórmula, un valor alto de *Merit* indica que las características están fuertemente correlacionadas con la variable dependiente (alta relevancia) y no son

muy redundantes entre si. Por lo contrario, un valor bajo de *Merit* indica que las características son muy redundantes entre sí y no están altamente correlacionadas con la variable dependiente.

```
from sklearn.feature_selection import mutual_info_classif

# Funcion para calcular el Merit
def calculate_merit_per_feature(X, y):
    merit_values = {}  # Diccionario para guardar los valores Merit
    n = X.shape[1]  # Numero de caracteristicas

    # Calcular la correlacion entre caracteristicas
    correlations_between_features = X.corr().abs()

    # Calcular la informacion mutua de cada caracteristica con la variable target
    mutual_info_values = mutual_info_classif(X, y)

    # Calcular el Merit para cada caracteristica
    for i, feature in enumerate(X.columns):
        avg_r_c = mutual_info_values[i]  # Informacion mutua entre la caracteristica y la variable target
        avg_r_r = correlations_between_features[feature].mean()  # Promedio de correlaciones con otras caracteristicas

        # Calculo del Merit para esta caracteristica
        merit = (n * avg_r_c) / np.sqrt(n + (n * (n - 1)) * avg_r_r)

        merit_values[feature] = merit

    return merit_values


# Calcular Merit
merit = calculate_merit_per_feature(X, y)
print(f"Merit: {merit}")
```

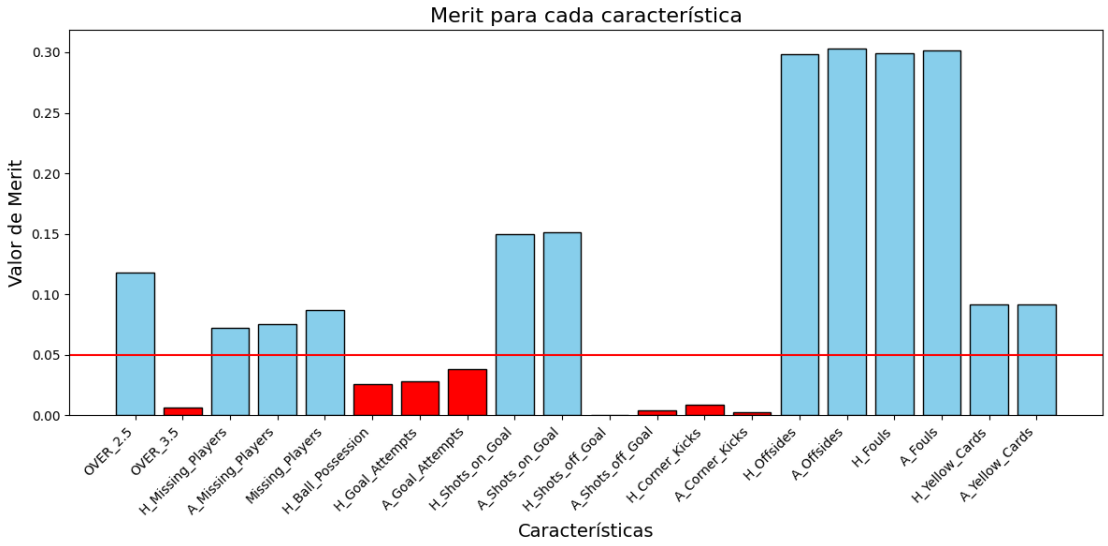

Figura 12: Valores de merito de cada parametro predictor.

En este caso, podemos establecer un umbral (0.05 por ejemplo), y trabajar unicamente con las variables que obtienen un valor de merito superior a este.

Por lo tanto, distintas técnicas de selección de variables dan lugar a distintos subconjuntos de parámetros predictores. En lugar de elegir uno u otro, vamos a trabajar con ambos conjuntos, y finalmente, nos quedaremos con el conjunto que mejor rendimiento obtenga a la hora de aplicar los modelos de clasificación.

3. Modelos de clasificación

Hasta ahora nos hemos dedicado a preparar los datos con el propósito de que los modelos sean lo más eficientes posible a la hora de clasificar nuevos datos. Ahora, una vez hemos aplicado todos los filtros y transformaciones que hemos considerado oportunas, vamos a definir nuestros conjuntos de datos para el entrenamiento y testeo de los modelos.

Por un lado, tenemos el conjunto de datos obtenido tras realizar la selección de variables mediante el test no paramétrico de *Kruskal-Wallis*. Por otro lado, tenemos un conjunto de datos más reducido, obtenido mediante la técnica de *Correlation Feature Selection*. Vamos a preparar estos conjuntos de datos para la aplicación de los modelos.

```
df1 = df_filtered_3.drop(columns = ['Missing_Players', '
    A_Corner_Kicks'])
df2 = df_filtered_3.drop(columns = ['OVER_3.5', '
    H_Ball_Possession', 'H_Goal_Attempts', 'A_Goal_Attempts',
     'H_Shots_off_Goal', 'A_Shots_off_Goal', 'H_Corner_Kicks'
    , 'A_Corner_Kicks'])
```

A continuación, debemos decidir como dividir los datos en los conjuntos de train y de test. Para ello, vamos a usar la técnica de *k-fold cross validation*, debido al considerable número de instancias que tenemos. Para esta técnica, se escoge un número entero k, y se divide el dataset en k subconjuntos del mismo tamaño. Una vez obtenidos los *k-folds*, realizamos k entrenamientos del modelo. En cada uno de estos, tomamos uno de los subconjuntos de datos como conjunto de test, y los $k-1$ restantes como conjunto de train. Finalmente, la media de las métricas de evaluación nos dirá si el modelo es adecuado o, si por lo contrario, debemos de ajustar algún hiperparametro o cambiar de arquitectura. En el caso de tener un resultado de evaluación bueno, realizaremos un entrenamiento final con todo el conjunto de datos, y ese será nuestro modelo.

```
from sklearn.model_selection import StratifiedKFold

kf = StratifiedKFold(n_splits=10, shuffle=True, random_state
    =42)

for train_index, test_index in kf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    # Aqui ajustas y evaluas el modelo
```

En nuestro caso, realizaremos particiones estratificadas de los datos, conservando los porcentajes de cada clase de la variable etiqueta tanto en el conjunto de entrenamiento como en el de test.

Ahora, es hora de escoger los modelos de clasificación. Empecemos con modelos simples, más fáciles de implementar y sobre todo de interpretar, los cuales suelen ser muy útiles para muchos de los problemas de clasificación que se plantean.

Primero, aplicaremos un *Árbol de Decisión*.

3.1. Decision tree

Los arboles de decisión son algoritmos que dividen el espacio en distintos fragmentos y a cada uno de estos le asignan, en tareas de clasificación, una clase de la variable etiqueta.

Estos algoritmos tienen estructura de árbol, y según bajamos en ellos, se va restringiendo el espacio mediante fronteras lineales. Por ejemplo, el primer nodo separa el espacio en dos, dada la condición $V1 < x$. Luego, en el segundo nivel del árbol, tendremos dos nodos, cada uno correspondiente al fragmento de espacio delimitado por la condición del nodo padre. En cada uno de estos nodos se establece una nueva frontera lineal, y por tanto, según bajamos en el árbol, mayores serán las restricciones y menor la entropía.

Durante el entrenamiento de los arboles de decisión, existen varios hiperparametros que definen el funcionamiento de estos algoritmos. Entre todos ellos, hemos escogido los siguientes:

- Profundidad máxima: Número máximo de niveles de un árbol, o lo que es lo mismo, número máximo de fronteras lineales que pueden delimitar el espacio de un nodo.
- Criterio de división: Criterio aplicado a la hora de establecer una nueva frontera en un nodo. Los criterios más empleados son:
    - Criterio de Gini: Mide que tan puras son las particiones resultantes. Trata de minimizar la siguiente métrica:

$$Gini = 1 - \sum_{i=1}^{n} p_i^2$$

    - Criterio de Entropía: Mide la incertidumbre en las particiones. Trata de minimizar la siguiente métrica:

$$H = -\sum_{i=1}^{n} p_i \cdot \log(p_i)$$

- Número de datos mínimo por nodo: Número de instancias mínimas para considerar la división del nodo.
- Número de datos mínimo por hoja: Número de instancias mínimas que puede tener una hoja.

En nuestro caso, establecemos ciertos intervalos para cada hiperparametro y generamos selecciones aleatorias de hiperparametros dentro de estos intervalos. Lo hacemos de manera aleatoria ya que consideramos que de esta manera se testan más valores para cada hiperparametro, comparando con la opción de usar una malla.

Finalmente, debemos escoger una métrica de evaluación. Escogemos el parámetro $F1_{weighted}$. Esta métrica considera la precisión y la exhaustividad para cada clase y calcula un promedio ponderado según el número de muestras en cada clase.

Para cada clase i, se calcula el *F1-score* como:

$$F1_i = \frac{2 \cdot (\text{precisión}_i \cdot \text{recall}_i)}{\text{precisión}_i + \text{recall}_i}$$

donde:

$$\text{precisión}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}, \quad \text{recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}$$

TP_i, FP_i, y FN_i son los valores verdaderos positivos, falsos positivos y falsos negativos de la clase i, respectivamente.

Cada *F1-score* se multiplica por el peso relativo de la clase, basado en el número de muestras en esa clase:

$$\text{weight}_i = \frac{\text{número de muestras en la clase } i}{\text{total de muestras}}$$

Finalmente, se suma el *F1-score* ponderado de todas las clases:

$$F1_{\text{weighted}} = \sum_i F1_i \cdot \text{weight}_i, \quad t.q. \quad 0 \leq F1_{\text{weighted}} \leq 1$$

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

for df in [df1, df2]:
    X = df.drop(columns=['Result'])
    y = df['Result']

    # Definir el clasificador y el espacio de busqueda
    clf = DecisionTreeClassifier(random_state=42, class_weight='balanced')
    param_dist = {
        'max_depth': randint(3, 20),  # Rango 3 a 20 parala profundidad maxima del arbol
        'min_samples_split': randint(10, 100),  # Rango de 2 a 20 para el minimo de muestras para dividir
        'min_samples_leaf': randint(5, 20),  # Rango de 1 a 20 para el minimo de muestras para la hoja
        'criterion': ['gini', 'entropy']
    }

    # Configurar GridSearchCV con validacion cruzada estratificada
    grid_search = RandomizedSearchCV(
        estimator=clf,
        param_distributions=param_dist,
        n_iter=20, # Numero de combinaciones aleatorias a probar
        scoring='f1_weighted',  # otras metricas como accuracy, precision, recall
        cv=StratifiedKFold(n_splits=10, shuffle=True, random_state=42),
        verbose=2,
        n_jobs=-1,  # Usar todos los nucleos disponibles
        random_state=42
    )
```

```
31
32      # Ajustar el GridSearch
33      grid_search.fit(X, y)
34
35      # Imprimir los mejores hiperparametros
36      print("Mejores hiperparametros encontrados:")
37      print(grid_search.best_params_)
38
39      # Obtener el mejor modelo
40      best_model = grid_search.best_estimator_
41
42      # Evaluar el modelo con la validacion cruzada original
43      y_pred_total = best_model.predict(X)
44      print("\nReporte de clasificacion final:")
45      print(classification_report(y, y_pred_total))
```

Mejores hiperparámetros encontrados:

- criterion = 'entropy'
- max_depth = 10
- min_samples_leaf = 8
- min_samples_split = 11

Reporte de clasificación final: df1

Clase	Precisión	Recall	F1-score
1	0.79	0.61	0.69
2	0.66	0.61	0.63
x	0.49	0.72	0.59

Reporte de clasificación final: df2

Clase	Precisión	Recall	F1-score
1	0.79	0.58	0.67
2	0.61	0.64	0.63
x	0.49	0.48	0.57

Como podemos observar, los resultados no son muy positivos. Aunque muy similares, los resultados con el primer dataset son algo mejores que con el segundo. Analicemos sobre este primer conjunto de datos que ha sucedido.

Para la clase '1' obtenemos una precisión de 0.79 y una exhaustividad de 0.61. Esto significa que de los datos que clasifica como clase '1' el 79 % están bien clasificados, pero de los datos que son realmente clase '1' solo el 61 % estan bien clasificados. Con la clase '2', la precisión baja considerablemente mientras que la exhaustividad sube. Lo mismo ocurre con la clase 'x', aun en mayor medida, llegando a una precisión inferior al 50 % y una exhaustividad del 68 %. En este caso, parece que se están asignando demasiados datos a las clases minoritarias, de ahí una precisión menor y exhaustividad mayor. Esto puede deberse a la influencia de los pesos en la métrica $F1_{\text{weighted}}$, la cual se emplea en casos de clases desbalanceadas. Probemos ahora una métrica de evaluación más sencilla, el *Accuracy*. Esta mide la proporción de predicciones correctas realizadas por el modelo en comparación con el total de predicciones. Su fórmula es:

$$\text{Accuracy} = \frac{\text{Número de predicciones correctas}}{\text{Número total de predicciones}}$$

Reporte de clasificación final: df1

Clase	Precisión	Recall	F1-score
1	0.82	0.60	0.69
2	0.68	0.60	0.64
x	0.49	0.78	0.60

Accuracy final: 0.6465

Reporte de clasificación final: df2

Clase	Precisión	Recall	F1-score
1	0.78	0.59	0.67
2	0.62	0.63	0.62
x	0.49	0.68	0.57

Accuracy final: 0.6240

Como vemos, los resultados son realmente similares. Dado el gran volumen de datos y lo repartida que esta la varianza entre las múltiples variables predictoras,

vamos a probar un modelo más sofisticado: *Random Forest*.

3.2. Random Forest

Este algoritmo es una combinación de múltiples arboles de decisión. Cada árbol se construye partiendo de un subconjunto distinto de instancias y características, y por ello, se generan arboles muy diversos. Para la selección de instancias, se hace uso de la técnica de *boostrapping*, la cual toma muestras aleatorias dentro del conjunto de datos original. Para los parámetros predictores, se seleccionan aleatoriamente por nodo ciertas variables para realizar la búsqueda de la restricción lineal optima.

Debido al dinamismo de los arboles de decisión, estos cambios en los datos generan arboles totalmente diferentes los unos de los otros, lo cual ayuda a la hora de tener en cuenta los distintos aspectos de nuestros datos, sin centrarnos en exceso en ciertas características. Finalmente, en nuestro caso, cada árbol devuelve una de las clases de nuestra variable etiqueta. Por ello, se escoge la moda para realizar una única predicción por instancia.

En este caso, hemos añadido dos nuevos hiperparametros a los ya empleados en el algoritmo anterior:

- Número de estimadores: Número de arboles empleados. A mayor número, mayor robustez pero mayor coste computacional
- Número máximo de parámetros: Número de parámetros que se consideran a la hora de escoger el parámetro óptimo para establecer una nueva frontera lineal en cada nodo. Al establecer un límite, forzamos a que cada nodo de cada árbol considere distintos parámetros para establecer una nueva restricción, y por lo tanto, aumentamos la variación entre los arboles.

```
1 from sklearn.ensemble import RandomForestClassifier
2
3 for df in [df1, df2]:
4     # Separar caracteristicas y etiqueta
5     X = df.drop(columns=['Result'])
6     y = df['Result']
7
```

```
    # Definir el clasificador y el espacio de busqueda
    clf = RandomForestClassifier(random_state=42,
        class_weight='balanced')
    param_dist = {
        'n_estimators': randint(50,300),  # Numero de
            arboles en el bosque
        'max_depth': randint(3, 20),  # Profundidad maxima
            del arbol
        'min_samples_split': randint(2, 20),  # Muestras
            minimas para dividir un nodo
        'min_samples_leaf': randint(1, 10),  # Muestras
            minimas en una hoja
        'criterion': ['gini', 'entropy'],  # Funcion de
            calidad del split
        'max_features': ['sqrt', 'log2', None]  # Seleccion
            de caracteristicas
    }

    # Configurar RandomizedSearchCV con validacion cruzada
        estratificada
    grid_search = RandomizedSearchCV(
        estimator=clf,
        param_distributions=param_dist,
        n_iter=10,  # Numero de combinaciones aleatorias a
            probar
        scoring='accuracy',  # Puedes usar otras metricas
            como precision, recall, f1
        cv=StratifiedKFold(n_splits=5, shuffle=True,
            random_state=42),
        verbose=3,
        n_jobs=-1,  # Usar todos los nucleos disponibles
        random_state=40
    )

    # Ajustar el RandomizedSearch
    grid_search.fit(X, y)

    # Imprimir los mejores hiperparametros
    print("Mejores hiperparametros encontrados:")
    print(grid_search.best_params_)
```

```
37
38      # Obtener el mejor modelo
39      best_model = grid_search.best_estimator_
40
41      # Evaluar el modelo con los datos de entrenamiento
            originales
42      y_pred_total = best_model.predict(X)
43      print("\nReporte de clasificacion final:")
44      print(classification_report(y, y_pred_total))
45
46      accuracy = accuracy_score(y, y_pred_total)
47      print(f"\nAccuracy final: {accuracy:.4f}")
```

Reporte de clasificación final: df1

Clase	Precisión	Recall	F1-score
1	0.95	0.84	0.89
2	0.90	0.87	0.89
x	0.79	0.85	0.86

Accuracy final: 0.8814

Reporte de clasificación final: df2

Clase	Precisión	Recall	F1-score
1	0.87	0.75	0.81
2	0.78	0.77	0.78
x	0.65	0.82	0.73

Accuracy final: 0.7743

En este caso, obtenemos resultados mucho mejores que en el caso anterior donde solo empleábamos un único árbol. Los resultados con el conjunto de datos más extenso son significativamente mejores, por lo que vamos a analizarlos.

Por un lado, seguimos viendo la misma tendencia de una precisión más alta en la clase '1', la que más instancia posee, y una exhaustividad mayor según baja la cantidad de datos de la clase, por lo que parece que el algoritmo sigue asignando más valores de los debidos a las clases minoritarias. Sin embargo, en este caso los resultados son notablemente mejores, con un accuracy final cercano al 90 %.

En este caso, los hiperparametros que mejor rendimiento han obtenido son algo diferentes a los empleados en el modelo de un único árbol.

Para empezar, se usa la entropía como métrica a la hora de establecer las fronteras lineales en cada nodo. Además, se crean arboles mucho más profundos, con un menor número de instancias limite para dividir los nodos. Esto parece indicarnos que al tener, en este caso, 120 arboles en nuestro algoritmo, podemos generar arboles más profundos que capten características más especificas, y luego, al juntar todos los arboles, obtenemos un modelo que generaliza bien.

Vamos a ver si podemos mejorar estos resultados con uno de los modelos de Machine Learning más populares hoy en día: *XGBoost*.

3.3. XGBoost

Este algoritmo también se basa en los arboles de decisión, y al igual que el *Random Forest*, hace uso de una amplia colección de ellos. Sin embargo, la gran diferencia entre el *Random Forest* y el *XGBoost* es que en este último, se aplica la técnica del descenso por gradiente de un árbol al siguiente, y así se trata de minimizar los errores cometidos.

De esta manera, el primer árbol se enfoca en ajustar las predicciones de manera general, y a partir de este, los demás arboles se centran en mejorar los errores del árbol anterior. Durante este proceso, el algoritmo minimiza la función de perdida escogida (*Categorical cross-entropy* en nuestro caso) mediante el descenso por gradiente, y de esta manera se ajustan los parámetros del siguiente árbol.

$$L = -\sum_{i=1}^{k} y_i \log(p_i)$$

Donde:

- y_i: es un valor binario (1 si pertenece a la clase i, 0 en caso contrario).
- p_i: es la probabilidad predicha por el modelo para la clase i.
- k: es el número total de clases.

Por todo esto, son muchos los hiperparametros que se pueden ajustar en este

tipo de modelo. Nosotros, por razones de tiempo y capacidad de computo, nos ceñiremos a los siguientes:

- Número de estimadores: Número de árboles en el modelo. Un número más alto generalmente mejora el rendimiento, pero también incrementa el tiempo de entrenamiento y la posibilidad de sobreajuste.
- Profundidad máxima: Profundidad máxima de cada árbol. Controla como de complejo puede ser el árbol. Árboles más profundos pueden modelar relaciones más complejas pero también tienden a sobreajustarse a los datos.
- Número de datos mínimo por hoja: Un valor más alto puede ayudar a evitar el sobreajuste, ya que garantiza que no se formen nodos con pocos datos.
- Learning_rate: Tasa de aprendizaje. Controla la magnitud de las actualizaciones durante el entrenamiento. Un valor más bajo mejora la generalización, pero requiere más árboles para su convergencia.
- Subsample: Proporción de muestras de entrenamiento utilizadas en cada árbol. Ayuda a prevenir el sobreajuste si se utiliza un valor menor a 1, ya que introduce aleatoriedad al modelo.
- Colsample_bytree: Proporción de características utilizadas para cada árbol. Ayuda a reducir el sobreajuste y a mejorar la generalización al reducir la correlación entre los árboles.
- Función objetivo: Función que especifica el tipo de problema que estamos resolviendo. En este caso, utilizamos *multi:softmax* para clasificación multiclase, donde el modelo predice la clase más probable para cada observación.

```
import xgboost as xgb
from sklearn.model_selection import RandomizedSearchCV,
   StratifiedKFold
from sklearn.metrics import classification_report,
   accuracy_score
from scipy.stats import randint

for df in [df1, df2]:
    # Separar caracteristicas y etiqueta
    X = df.drop(columns=['Result'])
```

```
    y = df['Result']
    y = y.replace({'x': 1, '1': 0, '2': 2})


    # Definir el clasificador y el espacio de busqueda
    clf = xgb.XGBClassifier(random_state=42,
       scale_pos_weight='balanced')
    param_dist = {
        'n_estimators': randint(50, 300),  # Numero de
           arboles en el bosque
        'max_depth': randint(3, 20),  # Profundidad maxima
           del arbol
        'min_child_weight': randint(1, 10),  # Minimo de
           instancias en el nodo hoja
        'learning_rate': [0.001, 0.01, 0.1, 0.2, 0.3],  #
           Tasa de aprendizaje
        'subsample': [0.7, 0.8, 0.9, 1],  # Proporcion de
           datos a usar en cada arbol
        'colsample_bytree': [0.7, 0.8, 0.9, 1],  #
           Proporcion de caracteristicas a usar en cada
           arbol
        'objective': ['multi:softmax']  # Funcion objetivo (
           dependiendo del tipo de problema)
    }

    # Configurar RandomizedSearchCV con validacion cruzada
       estratificada
    grid_search = RandomizedSearchCV(
        estimator=clf,
        param_distributions=param_dist,
        n_iter=50,  # Numero de combinaciones aleatorias a
           probar
        scoring='accuracy',  # Puedes usar otras metricas
           como precision, recall, f1
        cv=StratifiedKFold(n_splits=10, shuffle=True,
           random_state=42),
        verbose=3,
        n_jobs=-1,  # Usar todos los nucleos disponibles
        random_state=40
    )
```

```
36
37      # Ajustar el RandomizedSearch
38      grid_search.fit(X, y)
39
40      # Imprimir los mejores hiperparametros
41      print("Mejores hiperparametros encontrados:")
42      print(grid_search.best_params_)
43
44      # Obtener el mejor modelo
45      best_model = grid_search.best_estimator_
46
47      # Evaluar el modelo con los datos de entrenamiento
           originales
48      y_pred_total = best_model.predict(X)
49      print("\nReporte de clasificacion final:")
50      print(classification_report(y, y_pred_total))
51
52      accuracy = accuracy_score(y, y_pred_total)
53      print(f"\nAccuracy final: {accuracy:.4f}")
```

Reporte de clasificación final: df1

Clase	Precisión	Recall	F1-score
1	0.75	0.80	0.77
2	0.60	0.60	0.60
x	0.73	0.65	0.69

Accuracy final: 0.7039

Reporte de clasificación final: df2

Clase	Precisión	Recall	F1-score
1	0.71	0.78	0.74
2	0.57	0.57	0.57
x	0.71	0.61	0.65

Accuracy final: 0.6710

Como vemos, los resultados son peores que los anteriores. Esto puede deberse a que este algoritmo es mucho más sensible a los hiperparametros, y por limita-

ciones computacionales en este análisis, no podemos analizar profundamente las distintas combinaciones de estos. Además, estos modelos pueden tender al sobreajuste, ya que son modelos más complejos y detectan características de los datos más sofisticadas.

Por todo ello, nuestro modelo final estará compuesto por el algoritmo *Random Forest*, el conjunto de parámetros asociado al test de *Kruskal-Wallis* y los hiperparametros:

- criterion = 'entropy'
- max_depth = 17
- max_features = 'log2'
- min_samples_leaf = 4
- min_samples_split = 4
- n_estimators = 120

4. Interpretabilidad del modelo

Dependiendo del objetivo final del análisis, en muchos casos el trabajo podría finalizar aquí. Una vez obtenido un modelo que funciona relativamente bien, que clasifica los datos en la clase correcta, el trabajo esta prácticamente hecho si lo que buscamos es clasificar nuevos datos no etiquetados.

Sin embargo, como ya lo explicamos al comienzo de este texto, nuestro objetivo no es clasificar datos nuevos, ni siquiera predecir resultados de partidos futuros. Nuestro propósito en este análisis es entender que parámetros son los más relevantes en el resultado final del partido. Por tanto, una vez obtenido un modelo correcto, es hora de interpretarlo, de entender porque clasifica *'1'* un partido y *'x'* otro, de analizar cuales son los parámetros que se tienen en cuenta a la hora de establecer estas etiquetas.

Para ello, vamos a emplear dos técnicas distintas. Comencemos con la técnica de *Permutation Feature Importance*.

4.1. Permutation Feature Importance

Esta métrica, aunque no proporcione información sobre la relación entre los parámetros predictores y las diferentes clases, es muy fácil de interpretar.

Para empezar, se entrena el modelo con los datos originales, y se obtiene un resultado de evaluación (Accuracy en nuestro caso). Luego, se escoge un parámetro y se aplica una permutación aleatoria a los datos de dicho parámetro. Este nuevo conjunto de datos se vuelve a introducir en el modelo, y de este obtenemos una nuevo resultado de evaluación. Finalmente, la importancia de la característica escogida se calcula mediante la diferencia entre la primera evaluación y la segunda, ya que cuanto más cambie la evaluación, mayor habrá sido el impacto de la reordenación del parámetro, y por lo tanto, mayor será la importancia de este. Este proceso se repite varias veces, quedándonos finalmente con la importancia media.

$$Importancia = Rendimiento\_base - rendimiento\_permutado$$

```
from sklearn.inspection import permutation_importance

# Calcular la importancia de caracteristicas basada en
    permutacion
result = permutation_importance(best_model_rf, X, y,
    n_repeats=10, random_state=42, scoring='accuracy')

# Ordenar caracteristicas por su importancia media
sorted_idx = result.importances_mean.argsort()[::-1]
features = X.columns[sorted_idx]
importances_mean = result.importances_mean[sorted_idx]
importances_std = result.importances_std[sorted_idx]

# Crear el grafico de barras horizontales con error
plt.figure(figsize=(10, 6))
plt.barh(range(len(features)), importances_mean, xerr=
    importances_std, align='center', color='skyblue', ecolor=
    'black', capsize=5)
plt.yticks(range(len(features)), features)
plt.gca().invert_yaxis()  # Invertir para que la
    caracteristica mas importante este arriba
```

```
plt.xlabel("Importancia Media")
plt.title("Importancia de las Caracteristicas con Barras de Error")
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.show()
```

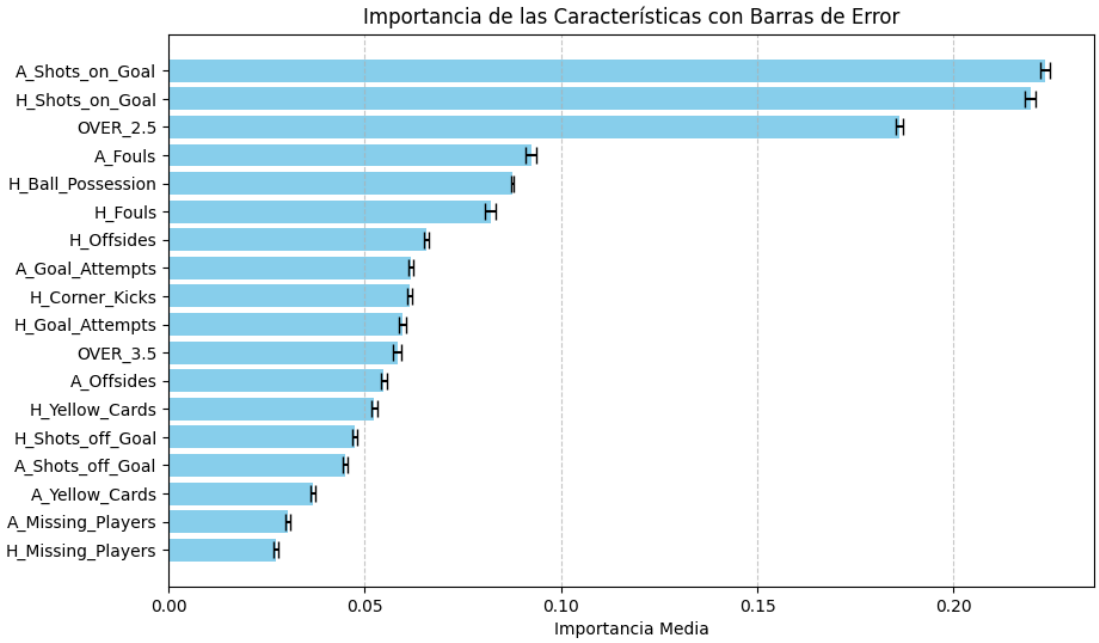

Figura 13: Valores de *Feature Importance* de cada parametro predictor.

Como podemos observar en el este gráfico, los parámetros más significativos son los tiros a puerta. Tras estos, el parámetro *OVER_2.5*, indicador de si hay más de 2 goles por encuentro, tiene también una alta relevancia en el modelo. A partir de ahí, las faltas de ambos equipos y la posesión de balón son otros de los parámetros que más afectan en la clasificación.

Aunque no nos resulta raro ver las métricas de tiros a puerta en primera posición, si que es sorprendente que el número de oportunidades de gol no tenga más impacto. Esto puede deberse a que, en el futbol, es más importante la calidad que la cantidad en lo que respecta a las ocasiones de gol.

No es para nada extraño ver equipos que controlan el juego, atacan continuamente sin demasiado peligro y pocas veces llegan a rematar a portería. Por ello, estos resultados nos invitan a darle mayor importancia al disparo a puerta, a la finalización de jugadas. De todas formas, los parámetros de disparos fuera son de

los menos importantes, por lo que no debemos recomendar el disparar en cualquier situación de juego, sino en aquellas con ciertas garantías de que al menos los disparos irán dirigidos a portería.

Obviamente el juego es mucho más complejo que esto, y deberían analizarse muchas más casuísticas, pero dada la dimensión de este análisis, creo que estas son interpretaciones congruentes con los resultados mostrados.

Vamos a contrastar estos resultados con una nueva métrica, quizá la más popular en la actualidad: *SHAP Values*.

4.2. SHAP Values

Los SHAP Values son una métrica que se calcula considerando las posibles conbinaciones de parámetros predictores y evaluando su contribución al modelo final. Proviene de la rama de Teoría de Juegos, y se calcula a partir de los valores de *Shapley*:

$$\phi_i(f) = \sum_{S \subseteq N \setminus \{i\}} \frac{|N|!}{|S|!(|N| - |S| - 1)!} \left[f(S \cup \{x_i\}) - f(S) \right]$$

Donde:

- N es el conjunto total de características.
- S es un subconjunto de características que no incluye a x_i.
- $f(S)$ es la predicción del modelo cuando se utiliza el subconjunto S de características.
- $f(S \cup \{x_i\})$ es la predicción del modelo cuando se utiliza el subconjunto S más la característica x_i.
- $\frac{|N|!}{|S|!(|N|-|S|-1)!}$ es un factor de ponderación que ajusta la importancia de cada subconjunto de características.

Mediante este análisis, obtenemos un gráfico para cada clase de nuestra variable etiqueta. De esta forma, esta técnica nos permite ver que impacto tiene cada

variable predictora en la predicción de cada clase, cosa que con el análisis previo nos era imposible.

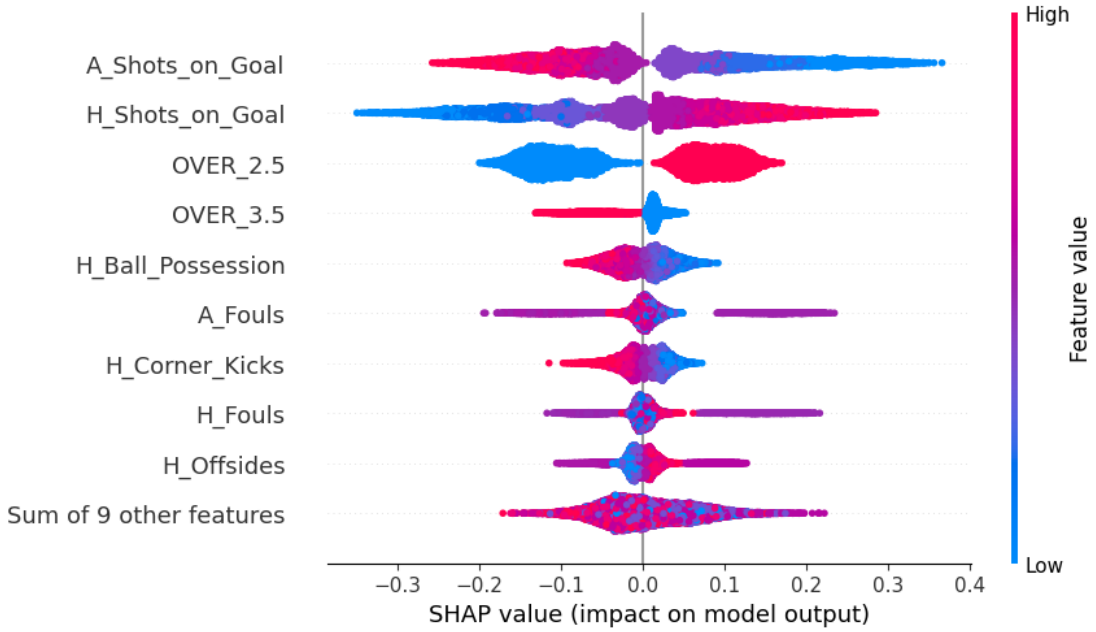

Figura 14: Valores de *Shap Values sobre la clase '1' (Victoria local).*

Para las victorias locales, observamos como los parámetros de tiros a portería tienen un gran impacto. Como era de esperar, los tiros a puerta del equipo visitante *A_Shots_on_Goal* contribuyen negativamente a la clasificación de la clase *Victoria Local*. Por lo contrario los tiros a puerta del equipo local *H_Shots_on_Goal* contribuyen positivamente.

Cabe destacar la siguiente variable con mas relevancia en la clasificación de la clase *1*: *OVER_2.5*. Esta tiene un impacto positivo, no muy alto, pero claro. Los partidos con mas de 2 goles tienden a clasificarse como victorias locales. Por el contrario, los partidos con mas de 3 goles *OVER_3.5*, muestran una tendencia totalmente contraria.

Además, es curioso como la posesión de balón del equipo local esta ligeramente relacionada positivamente con la no victoria local, lo cual no podemos considerar erróneo considerando que no siempre el que domina el balón acaba superando al rival.

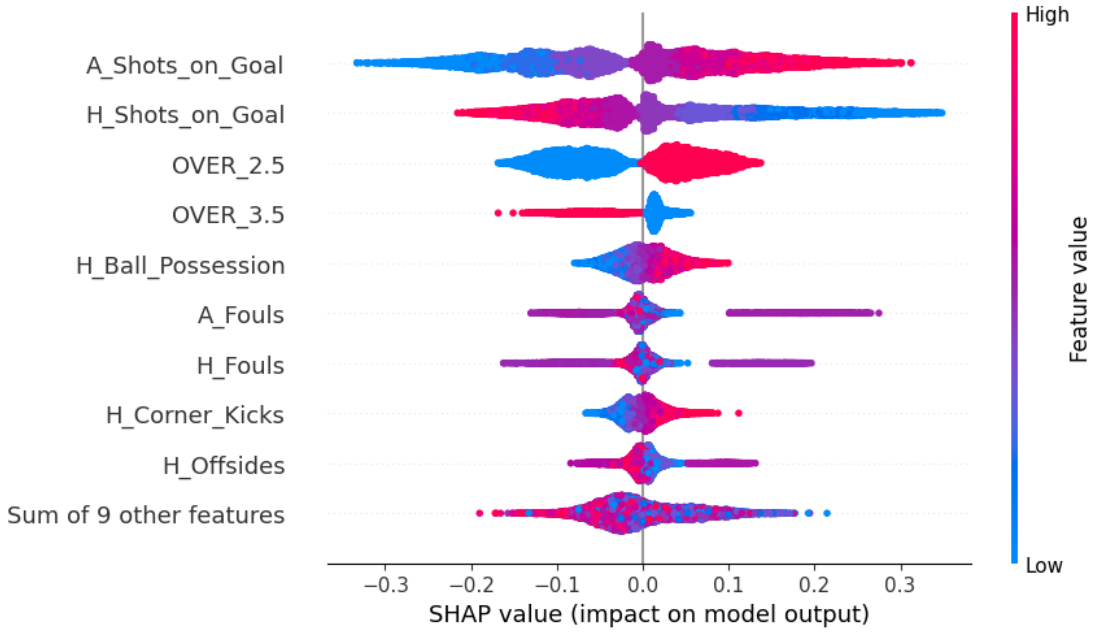

Figura 15: Valores de *Shap Values sobre la clase '2' (Victoria Visitante).*

Los resultados para esta segunda clase, *Victoria Visitante*, son lógicamente muy distintos a los anteriores. En concreto, son resultados completamente opuestos, lo cual es lógico tratándose del resultado antagónico al anterior. Sin embargo, vemos que esta diferencia no se da en las variables referentes al número de goles *OVER_2.5* y *OVER_3.5*. En esto casos, las variables siguen teniendo el mismo impacto que en la primera clase, lo cual nos hace pensar que son variables que el modelo usa para distinguir las clases *1* y *2* de la clase *x*. Comprobemoslo.

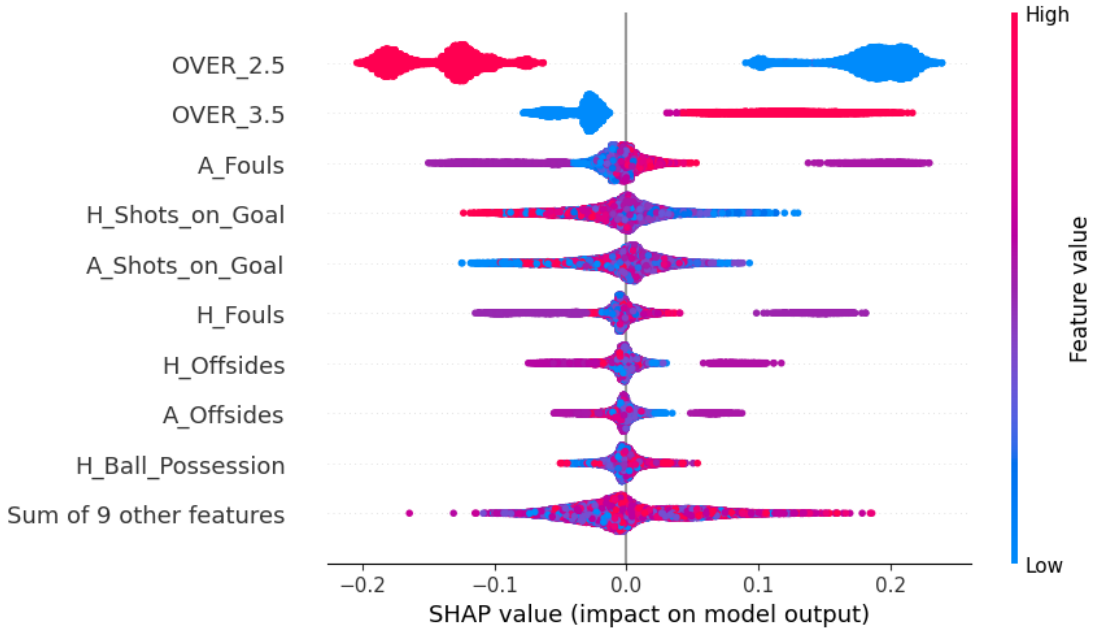

Figura 16: Valores de *Shap Values sobre la clase 'x' (Empate).*

Como predecíamos, los parámetros *OVER_2.5* y *OVER_3.5* son los más importantes a la hora de clasificar un partido como empate. Vemos que los partidos con más de 2 tienden a no clasificarse como empate, mientras que los partidos con más de 3 goles si. Esto se puede interpretar de la siguiente manera:

Los partidos con 2 goles o más son partidos más abiertos, con mayor abanico de resultados posibles, y por lo tanto, estilísticamente es más improbable que el resultado final sea de empate. Sin embargo, los partidos con más de 3 goles, aunque debieran de seguir la misma lógica, no son muy abundantes. Por ende, puede ser que no se este interpretando correctamente esta variable.

Con todo esto dicho, podríamos sacar muchos resultados, pero creemos que las tres conclusiones principales de este análisis son:

- El parámetro que más impacto tiene en el resultado final del partido es el número de tiros a puerta. Cuantos más tiros a puerta realice un equipo, mayor será la probabilidad de victoria.
- La posesión de balón no tiene un impacto positivo en el resultado final. Es más, parece que tener una alta posesión puede tener un leve impacto negativo en las probabilidades de victoria del equipo.

- Los partidos con mayor número de goles suelen decantarse con victoria local o visitante, no con empate

5. Comparación estadística de modelos

Para finalizar con el análisis, vamos a comparar estadísticamente los modelos aplicados a los dos conjuntos de datos. De esta forma, podremos justificar estadísticamente la elección final de nuestro modelo. Esto podría no parecer necesario, ya que ya hemos hecho de cierto modo una comparativa entre los modelos, haciendo uso de la métrica de evaluación *Accuracy*. Sin embargo, esta métrica es el resultado de hacer la media de las distintas evaluaciones en el proceso de entrenamiento, y por tanto, no estamos teniendo en cuenta factores como la consistencia del modelo. Por tanto, vamos a comparar los errores de nuestros modelos con el objetivo de fundamentar nuestra elección final.

Primero, recogemos los resultados de los 3 modelos aplicados a los dos conjuntos de datos.

Accuracy de los distintos modelos sobre los conjuntos de datos

Decision Tree	**Random Forest**	**XGBoost**
0.6465	0.8814	0.7039
0.6240	0.7743	0.6710

Luego, aplicamos el *test de Friedman*:

- H_0: Todos los clasificadores tienen un rendimiento similar.
- H_1: Existe al menos un par de clasificadores con rendimientos significativamente diferentes.

El estadístico empleado para este test se calcula:

$$Q = \frac{12}{n \cdot k(k+1)} \sum_{j=1}^{k} R_j^2 - 3n(k+1)$$

donde:

- n: Número de bloques (filas).
- k: Número de métodos (columnas).
- R_j: Suma de los rangos del método j. $R_j = \sum_{i=1}^{n} r_{ij}$
- r_{ij} es el rango del método j en el bloque i

En el caso de obtener un p-valor inferior a 0.05, podremos rechazar la hipótesis nula, y por lo tanto, suponer que existen diferencias significativas entre al menos dos modelos. En ese caso, se realizara un segundo test, el test de Nemenyi, para identificar diferencias entre pares de modelos.

```
from scipy.stats import friedmanchisquare
# pip install scikit-posthocs
import scikit_posthocs as sp

stat, p_value = friedmanchisquare(*[df[col] for col in df.
    columns])

print(f'Estad stico de Friedman: {stat}')
print(f'Valor p: {p_value}')

if p_value < 0.05:
    print("Hay diferencias significativas entre los modelos.
        ")

    nemenyi_result = sp.posthoc_nemenyi_friedman(df.values)
    print(nemenyi_result)

else:
    print("No hay diferencias significativas entre los
        modelos.")
```

Los resultados del test de Friedman (p-valor = 0.135) nos dicen que no existen diferencias significativas entre los modelos. Aun así, como estos resultados van en contra de lo que nos dice la intuición, con una diferencia tan grande en los niveles de accuracy de los modelos, vamos a comparar dos de los modelos empleados, y veamos si esta comparativa nos da un resultado diferente.

Para ello, primero comprobaremos si la diferencia entre los errores obtenidos con ambos modelos sigue una distribución normal, mediante el test de *Shapiro-Wilk*.

- H_0: La muestra proviene de una población con distribución normal.
- H_1: La muestra no proviene de una población con distribución normal.

Con el estadístico:

$$W = \frac{\left(\sum_{i=1}^{n} a_i x_{(i)}\right)^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Donde:

- n: Tamaño de la muestra.
- $x_{(i)}$: Datos ordenados de menor a mayor.
- $\bar{x}$: Media muestral.
- a_i: Coeficientes calculados a partir de los valores esperados y la matriz de covarianza de una distribución normal.

```
from scipy.stats import shapiro
import matplotlib.pyplot as plt
# Calcular diferencias
differences = np.array(errors_per_fold_dt) - np.array(
    errors_per_fold_rf)

# Test de normalidad: Shapiro-Wilk
shapiro_stat, shapiro_p = shapiro(differences)
```

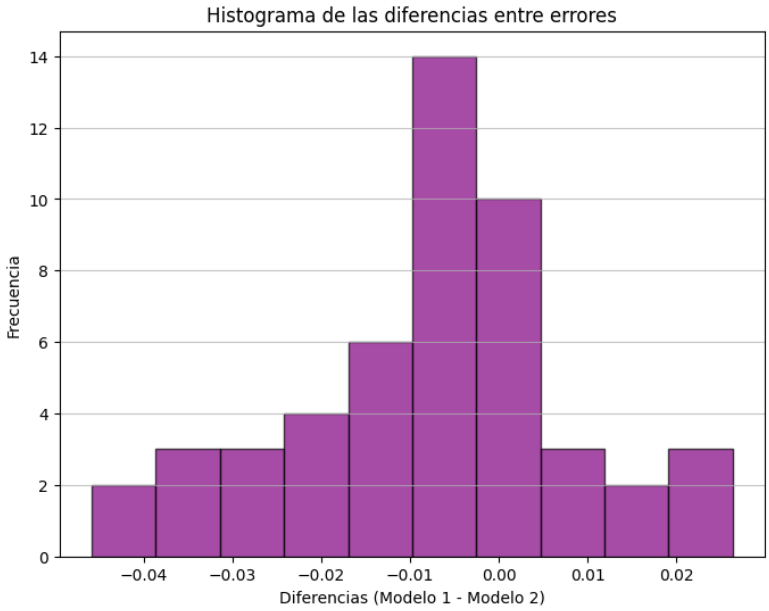

Figura 17: Valores de *Diferencias entre errores de los algoritmos 'Decision Tree' y 'Random Forest'.*

El p-valor es mayor a 0.05 (0.559), y por lo tanto, rechazamos la hipótesis nula. En consecuencia, podemos asumir normalidad en nuestros datos.

Por ello, podemos realizar un *t-test* para comparar ambos modelos.

El test *t de Student* es una prueba paramétrica que se usa para comparar las medias de dos grupos independientes, evaluando si son significativamente diferentes.

- H_0: Las medias de las dos muestras son iguales. Es decir, no hay diferencia significativa entre los grupos.
- H_1: Las medias de las dos muestras son diferentes. Es decir, hay una diferencia significativa entre los grupos.

El estadístico t se calcula de la siguiente manera:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

donde:

- $\bar{X}_1$ y $\bar{X}_2$ son las medias de las dos muestras.
- s_1^2 y s_2^2 son las varianzas de las dos muestras.
- n_1 y n_2 son los tamaños de las dos muestras.

```
from scipy.stats import ttest_ind

# Prueba t para muestras independientes
stat, p_value = ttest_ind(errors_per_fold_dt,
    errors_per_fold_rf, alternative='two-sided')
```

Según el resultado del t-test, con un p-valor de 0.013, podemos asumir que los resultados de los modelos son significativamente diferentes.

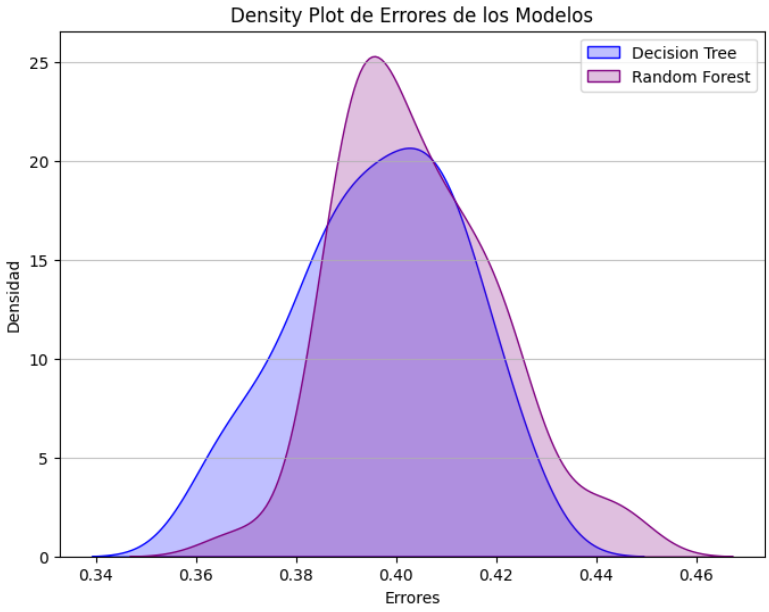

Figura 18: *Distribución de los errores de los algoritmos 'Decision Tree' y 'Random Forest'.*

Por todo ello, concluimos que realmente si que hay una diferencia real entre ambos modelos aplicados sobre el mismo conjunto de datos. Sin embargo, estos resultados nos indican que es de vital importancia ajustar bien los hiperparametros y precisar correctamente una buena rutina de entrenamiento y de validación a la hora de aplicar cualquier modelo clasificador a los datos.

6. Código

El código fuente de este texto puede ser consultado en el siguiente enlace: Github

Referencias

[1] Leo Breiman. Classification and regression trees. *CRC press*, 1986.

[2] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[3] M. M. Breunig, H. P. Kriegel, R. T. Ng, and J. Sander. Lof: Identifying density-based local outliers. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 93–104, 2000.

[4] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[5] M. Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

[6] W. H. Kruskal and W. A. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.

[7] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems*, 30, 2017.

[8] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[9] Student. The probable error of a mean. *Biometrika*, 6(1):1–25, 1908.

[10] I. K. Yeo and R. Johnson. A new family of power transformations to improve normality or symmetry. *Biometrika*, 87(4):954–959, 2000.