PRÁCTICA FINAL:
PREDICCIÓN DE PITCHES

Contents

Motivación

La MLB es una liga de béisbol de gran importancia debido a su historia, calidad del juego, relevancia cultural y mediática. Su estatus como una de las ligas más destacadas del mundo la convierte en un referente importante para los amantes del béisbol y un motor clave dentro de la industria deportiva. También la podemos ver desde el punto de vista económico, ya que según Forbes se estima que los equipos consiguieron en 2022 una cifra récord de entre $10.8 y $10.9 miles de millones en ingresos.

En un mundo tan competitivo, siempre se están buscando maneras de mejorar tus resultados con posibles ventajas sobre tus rivales. Además, es un deporte donde las estadísticas son una parte muy importante del juego. Es por ello por lo que se lleva varios años utilizando modelos analíticos predictivos para mejorar el rendimiento de los equipos. La película de *"Moneyball"*, basada en hechos reales, demuestra cómo es posible montar un buen equipo con poco presupuesto mediante modelos estadísticos.

Introducción

En beisbol, al lanzamiento que efectúa el pitcher al bateador se le denomina “pitch”. El pitcher intentará eliminar al bateador, bien por Strikes (no consigue contactar con la bola en 3 lanzamientos) o porque han cogido en el aire su bola bateada. El pitcher utilizará distintos tipos lanzamientos con tal de confundir al bateador, cambiando la trayectoria y la velocidad. Existen muchos tipos de pitches, pero en este proyecto destacaremos los 5 más usados:

- Fastball
- Slider
- Changeup
- Curveball
- Cutter

Objetivo

En nuestro proyecto contamos con millones de datos sobre todos los pitches lanzados entre 2015 y 2018 en la MLB. Existen campos como la velocidad en varias dimensiones, aceleración, posición, velocidad de giro, bateador…

Queremos clasificar qué tipo de pitch es en función de las características del lanzamiento. Este modelo podría tener 2 principales utilidades. Primero, para que en la retrasmisión por televisión podemos ver en cuestión de milisegundos el tipo de pitch lanzado con confianza sin que un experto tenga que analizarlo. Esto podría ayudarnos a tener mejores Ratings de audiencia. Por otro lado, se podría utilizar en los partidos, para ayudar a los bateadores a identificar el tipo de pitch que les viene para poder batearlo y tener más opciones de llegar a base a salvo.

Podemos fijar cómo objetivo principal poder clasificar el lanzamiento en uno de los 5 principales tipos con un acierto de al menos el 80 %.

Módulos del proyecto

Carga de los datos

Los datos vienen del Dataset de Kaggle https://www.kaggle.com/datasets/pschale/mlb-pitch-data-20152018?select=pitches.csv , contando con 40 columnas y 2867155 filas en lo que encontramos los datos de todos los Pitches lanzados entre 2015 y 2018.

Tenemos 3 archivos, en el principal (pitches.csv) tenemos los pitches lanzados, incluyendo los códigos de los jugadores, y de la jugada. Los datos de los jugadores y de las jugadas los encontramos en los archivos “player_names.csv” y “atbats.csv” respectivamente. Por lo tanto, para tener todo en una misma estructura, cargaremos los 3 dataframes y los uniremos con merge. Por si queremos hacer algún análisis por jugador, nos creamos una nueva columna con el nombre y el apellido.

```
pitches= pd.read_csv('pitches.csv')
atbats= pd.read_csv('atbats.csv')
player_name= pd.read_csv('player_names.csv')
pitches['ab_id'] = pitches['ab_id'].astype(int)
```

```
player_name.rename(columns={'id': 'batter_id'}, inplace=True)
data=pd.merge(pitches, atbats,  how='left', left_on='ab_id', right_on = 'ab_id')
data=pd.merge(data, player_name,  how='left', left_on='batter_id', right_on =
'batter_id')
data['Batters Name'] = data[['first_name', 'last_name']].apply(lambda x: '
'.join(x), axis=1)
data=data.drop(['first_name','last_name','code'],axis=1)
data.head()
```

Limpieza de datos

Queremos clasificar el tipo de pitch que se lanza en función de sus características, por lo tanto habrá muchas columnas que no serán relevantes para la clasificación y lo único que harán será causar un posible overfitting.

Estas columnas serán campos como el nombre de los jugadores, algún dato de la situación del partido, el conteo del bateador, etc. He eliminado las columnas basadas en mi criterio, ya que sigo desde hace años el mundo del beisbol y considero que no tienen ningún tipo de relación con la variable objetivo.

Estas son las 27 columnas eliminadas:

```
data=data.drop(['type','on_1b','on_2b','on_3b','pitch_num','outs','b_score','even
t_num','batter_id','p_score','p_throws','Batters
Name','stand','inning','top','type_confidence','ab_id','o','pitcher_id','event','
g_id','b_count','s_count','sz_bot','sz_top','px','pz'],axis=1)
```

Por lo que nos quedaremos con las siguientes 23 columnas:

Index(['start_speed', 'end_speed', 'spin_rate', 'spin_dir', 'break_angle', 'break_length', 'break_y', 'ax', 'ay', 'az', 'vx0', 'vy0', 'vz0', 'x','x0', 'y', 'y0', 'z0', 'pfx_x', 'pfx_z', 'nasty', 'zone'],dtype='object')

Estas columnas incluyen información como la posición del pitch, la aceleración o la velocidad en tres dimensiones, la velocidad de salida, el ángulo o la velocidad de giro.

Después de separar la variable target (¨Pitch Type¨) de las otras columnas, el siguiente paso será la normalización de las variables que vamos a utilizar. En este caso no tenemos variables categóricas que necesitan codificación.

A continuación se muestra la cabecera de los datos normalizados.

	start_speed	end_speed	spin_rate	spin_dir	break_angle	break_length	break_y	ax	ay	az	...	vz0	x	x0	y	y0	z0	pfx_x	pfx_z	nasty	zone
0	0.800347	0.784091	0.455084	0.468490	0.227778	0.125506	0.500	0.561688	0.603238	0.650464	...	0.378417	0.456025	0.370644	0.652946	0.0	0.571766	0.542155	0.723911	0.191489	0.307692
1	0.769097	0.787879	0.197967	0.513490	0.272840	0.218623	0.625	0.480953	0.444026	0.483252	...	0.399537	0.354566	0.262005	0.739030	0.0	0.540555	0.471278	0.577001	0.542553	1.000000
2	0.800347	0.803030	0.412033	0.502993	0.271605	0.133603	0.500	0.487575	0.523574	0.627627	...	0.358996	0.443197	0.340901	0.720269	0.0	0.508438	0.477279	0.700861	0.223404	0.307692
3	0.831597	0.835227	0.256217	0.468443	0.253395	0.178138	0.500	0.532727	0.551839	0.524929	...	0.448224	0.521887	0.259167	0.635585	0.0	0.558701	0.515004	0.604103	0.361702	0.000000
4	0.751736	0.750000	0.308117	0.494182	0.264198	0.190283	0.500	0.501801	0.520485	0.552708	...	0.356409	0.349996	0.336701	0.799832	0.0	0.475050	0.490140	0.646910	0.787234	1.000000

5 rows × 22 columns

Nos centraremos ahora en la variable objetivo. Aunque aparecen hasta 15 tipos de pitches diferentes, muchos de ellos son poco frecuentes y complicados de identificar, así que nos quedaremos con los 5 principales pitches que serán los más frecuentes. A cada pitch se le ha asignado un número entero de la siguiente manera:

- Fastball (FF)-> 1
- Slider (SL) -> 2
- Changeup (CH)-> 3
- Curveball (CU) -> 4
- Cutter (FC) -> 5

Además, he decidido recortar el número de filas a usar para los modelos. Me parece que, si usamos tantísimos datos, correr los modelos en local va a ser muy pesado y va a llevar mucho tiempo. No podremos además utilizar Grid Search para buscar los parámetros óptimos de modelos pesados como Random Forest, ya que podrían tardar muchas horas o incluso días.

Por lo tanto, en principio utilizaremos 300000 filas de datos, ya que considero que es una cantidad más que suficiente para entrenar un modelo preciso y a la vez los modelos pueden entrenarse en un tiempo relativamente moderado. Lo he elegido considerando que haré undersampling para combatir el class imbalance y por lo tanto acabaremos con menos datos.

Split test y train

Dividimos ahora los datos entre 2 datasets, uno para el entrenamiento y otro para el test. He decidido optar por el clásico 80-20 %, ya que nos permite tener bastantes datos para el entrenamiento y todavía quedarán suficientes para evaluar el modelo.

```
    print(len(X_train))
    print(len(X_test))
✓ 0.0s
240000
60000
```

Class Imbalance

Si vemos cuántos pitches tenemos de cada clase en el conjunto de entrenamiento habrá un claro desequilibrio:

```
1    115199
2     47107
3     34566
4     23914
5     19214
Name: pitch_type, dtype: int64
```

El más común será el Fastball, con más del doble que el siguiente, mientras que el menos frecuente será el Cutter con menos de 20000.

Normalmente en clase utilizamos SMOTE para combatir esta desigualdad, pero en este caso prefiero evitar generar muestras nuevas con SMOTE ya que pueden afectar a la accuracy del modelo. La diferencia es que ahora tenemos 300000 datos y aun haciendo undersampling a la clase menor, tendríamos cerca de 100000 , que deberían ser suficientes para entrenar un buen modelo.

Concretamente realizaré el undersampling con Near Miss. El algoritmo de NearMiss se basa en la idea de eliminar muestras de la clase mayoritaria que están "cerca" de las muestras de la clase minoritaria. El término "cerca" se refiere a la distancia entre las muestras de la clase mayoritaria y las muestras de la clase minoritaria en el espacio de características. De las 3 versiones que tiene, yo utilizaré la 1, en la que se seleccionarán las muestras cuyas distancias medias a las k instancias más cercanas de la clase minoritaria son las más pequeñas.

El proceso del algoritmo de NearMiss para no perder información realiza los siguientes pasos:

- Calcula la distancia entre las muestras de la clase mayoritaria y las muestras de la clase minoritaria utilizando una medida de distancia.
- Aplica una estrategia de selección basada en la distancia para determinar qué muestras de la clase mayoritaria se mantendrán en el conjunto de datos reducido.
- Combina las muestras que mantenemos de la clase mayoritaria con las muestras de la clase minoritaria para formar un conjunto de datos reducido y equilibrado.

El resultado que obtenemos será un dataset donde todas las clases tienen las mismas muestras que la clase minoritaria, 19214:

```
1    19214
2    19214
3    19214
4    19214
5    19214
dtype: int64
```

Modelos

El último módulo será el entrenamiento y validación de 4 algoritmos de clasificación:

- Decision Trees
- KNN neighbours
- Random Forests
- Multilayer Perceptron

Para cada uno se buscarán los parametros óptimos que permitan obtener el menor error posible. En todos los modelos se ha utilizado la librería de Python Scikit-learn. La implementación completa se detalla en el siguiente apartado.

Algoritmos y parámetros usados

Decision Trees

Los árboles de decisión son estructuras de flujo de control en forma de árbol donde cada nodo interno representa una característica o atributo, cada rama representa una regla de decisión basada en ese atributo, y cada hoja representa una etiqueta o valor de salida.

El proceso de construcción de un árbol de decisión implica seleccionar la mejor característica en cada nodo para dividir el conjunto de datos en subconjuntos más puros y homogéneos en términos de las etiquetas de salida. Esto se realiza de forma recursiva hasta que se cumplan ciertos criterios de parada, como alcanzar una profundidad máxima o no tener suficientes muestras en un nodo. Una vez construido el árbol, se puede utilizar para predecir la clase o valor de salida de nuevas instancias al seguir el camino correspondiente desde la raíz hasta una hoja.

He decido elegirlo como punto de partida por su fácil interpretabilidad, eficiencia computacional y robustez frente a datos ruidosos. Será un algoritmo rápido de entrenar y que ya nos dará una idea de cómo de efectiva puede ser la clasificación.

Lo primero que debemos hacer es elegir el conjunto de posibles parámetros para evaluar:

- El primero y más importante será ***max_depth***, que controlará la complejidad del árbol al fijar la profundidad máxima que puede tener. Si es muy grande, es posible que nuestro árbol pierda capacidad de generalización, mientras que un valor pequeño podrá resultar en una precisión insuficiente. Probaremos con distintos valores como 5,10,20 y None (Sin restricción)
- El siguiente es ***min_samples_split***, el mínimo número de muestras a considerar al crear un nodo. Un número muy pequeño causará overfitting al crearse nodos con pocas muestras, mientras que uno muy grande no permitirá un desarrollo completo del árbol. Elegimos 20,40,60 y 80.
- ***min_impurity_decrease*** especifica que un nodo se separará si la separación genera un descenso de la impureza mayor o igual a este valor. Controla por lo tanto el crecimiento del árbol y de posibles valores elegiremos números pequeños como 0.0001, 0.0005 y 0.001.
- El criterio para elegir separador se selecciona con ***criterion.*** Consideramos dos posibilidades: Gini (la usada en los árboles CART) y entropy (usada en ID3 y C4.5).
- Por último, ***max_features*** nos indica el número de variables a considerar cuando elegimos un separador. Consideramos 'auto','sqrt' y 'log2', estando este valor en función del número de variables total.

```
param_grid = {
    'max_depth':[5,10,20,None],
    'min_samples_split':[20,40,60,80],
    'min_impurity_decrease': [0.0001, 0.0005, 0.001],
    'criterion' :['gini', 'entropy'],
    'max_features': ['auto','sqrt', 'log2']
}
```

Finalmente, la búsqueda nos encuentra los siguientes parámetros óptimos:

Initial parameters: {'criterion': 'entropy', 'max_depth': None, 'max_features': 'auto', 'min_impurity_decrease': 0.0001, 'min_samples_split': 40}

score: 0.7962631414593525

La precisión conseguida con estos parámetros es de 79.63%. Por otro lado mostramos ahora la importancia de los predictores en el modelo.

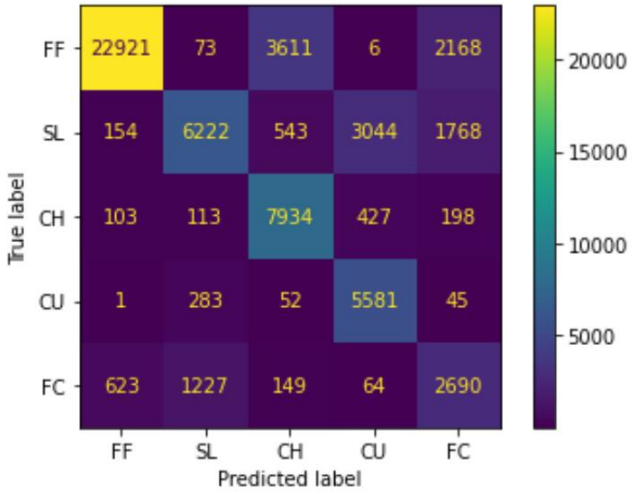
	predictor	importance
19	pfx_z	0.187409
11	vy0	0.146709
12	vz0	0.114867
5	break_length	0.112115
9	az	0.099370
18	pfx_x	0.067551
7	ax	0.047585
4	break_angle	0.041471
3	spin_dir	0.036695

El primero será pfx_z(altura del lanzamiento), seguido de vy0 y de vz0 (componentes de la velocidad en las direcciones Y y Z). No habrá un predictor muy dominante, sino que está bastante repartida la importancia, especialmente entre los 5 primeros.

Probando el modelo en el conjunto de test nos encontramos una precisión algo menor, alrededor de 75.58 %

ACCURACY OF THE MODEL: 0.7558

Por otro lado, la matriz de confusión nos muestra lo siguiente:



Tal y cómo habíamos visto en el conjunto de entrenamiento, lo más común será la Fastball (FF), que se clasificará correctamente un 79.6 % de las veces. Vemos además que hay muchas Fastballs que se clasifican incorrectamente como Changeup (CH), en un 12.5% de las ocasiones, o como Cutter (FC) en un 7.5 %. Otro gran problema de este modelo ocurrirá cuando tenemos una Slider (SL) ya se clasifica cómo Curveball (CU) incorrectamente un 25.91 % de veces. Por último, podemos destacar la confusión al distinguir entre Cutters vs Sliders, con un 25.81 % de Cutters predichos como Sliders. Los lanzamientos que mejor se clasificarán serán los Curveballs, con un 93.6 % de acierto.

En general de este modelo podemos decir que no tiene mala precisión, pero es bastante mejorable, se confunde al clasificar varios tipos de pitches, especialmente con el Slider.

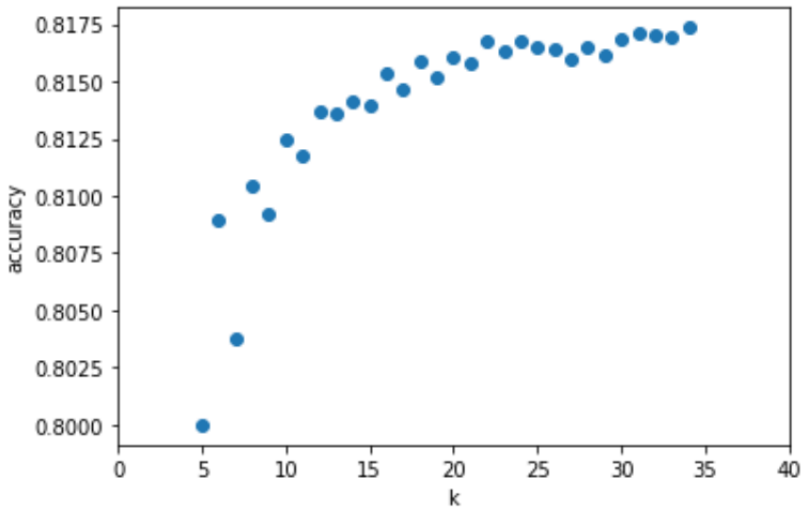
KNN vecinos

El funcionamiento del algoritmo K-Nearest Neighbors (KNN) se basa en la idea de que los objetos similares tienden a estar cerca unos de otros en un espacio de características. Dado un conjunto de datos de entrenamiento con etiquetas conocidas, el algoritmo KNN clasifica nuevos ejemplos basándose en la similitud con los K ejemplos de entrenamiento más cercanos.

Lo he seleccionado ya que es un algoritmo intuitivo y fácil de entender, con capacidad para manejar datos no lineales. Además, tiene muchos menos parámetros que los otros algoritmos, lo que lo hace más simple. El principal inconveniente es que tardará bastante más que el árbol de decisión, especialmente con un dataset relativamente grande como este, ya que tendrá que calcular muchas distancias.

Si dejamos la distancia como *Euclidea* por defecto, el parámetro más relevante que es necesario tunear es el número de vecinos a considerar. Este es muy importante ya que puede causar un overfitting o underfitting si es muy bajo o alto respectivamente.

Para seleccionarlo, probaremos a entrenar el modelo con un número de vecinos entre 5 y 34, registrando en la siguiente tabla, la precisión conseguida.



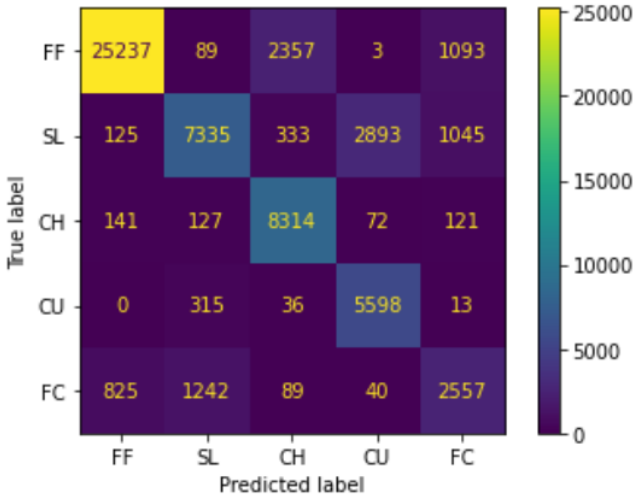
Se aprecia claramente como tenemos que coger 34 vecinos para el modelo. Una vez seleccionado este parámetro procedemos a entrenarlo y evaluarlo con el conjunto de test.

Se obtiene una precisión del 83.68 % en el entrenamiento y del 81.74 % en el test.

Accuracy of K-NN classifier on training set: 0.8368

Accuracy of K-NN classifier on test set: 0.8174

La matriz de confusion en el test será la siguiente:



Se aprecia de primeras como este es un predictor mucho mejor que el árbol de decisión. Por ejemplo para los lanzamientos que son Fastballs (FF), se clasifican correctamente el 87.7 %, comparado con el 79.6 % del árbol. Los clasificados incorrectamente como Changeup (CH) han bajado hasta el 8.2% y los Cutters (FC) hasta el 3.8%.

Sin embargo, seguimos teniendo un gran problema con los Sliders (SL), clasificándolos como Curveball (CU) un 24.7 %. Además, sigue habiendo una gran confusión entre Sliders y Cutters.

En general podemos decir que este es un predictor mucho mejor que el árbol de decisión, con un 82 % de precisión en test. Mejora especialmente la clasificación de las Fastballs, aunque sigue teniendo un problema con los Sliders.

Random Forest

Random Forest es un algoritmo de tipo *bagging*, donde combinaremos modelos débiles para obtener uno fuerte. En concreto, se pretende minimizar la varianza del modelo, es decir, cuánto cambian las predicciones si entrenamos el modelo con diferentes muestras de datos.

En este algoritmo, se deberán crear varios árboles de decisión, realizando una predicción en base a lo que han votado la mayoría de los árboles. Para reducir la correlación entre los árboles, entrenaremos cada uno con un conjunto de muestras distintas, seleccionando además solo una parte de las variables. Si no hiciésemos esto, la mayoría de árboles se desarrollaría basándose solo en los mejores predictores y estarían bastante correlados entre sí.

He decidido usar este algoritmo ya que, habiendo probado el árbol de decisión, me parecía natural seguir con una combinación de ellos como el Random Forest, que debería mejorar la precisión. Además, son resistentes al sobreajuste (overfitting) debido a la aleatoriedad introducida durante la construcción de los árboles.

El primer paso será elegir el conjunto de posibles parámetros para evaluar. La mayoría de ellos es común con el árbol de decisión:

- ***n_estimators*** indica cuántos árboles de decisión usaremos. Probaremos con 50,100 y 150.
- Tendremos ***max_depth***, que controlará la complejidad del árbol al fijar la profundidad máxima que puede tener. Probaremos con distintos valores como 3,10,20 y None (Sin restricción)
- El criterio para elegir separador encada uno de los árboles se selecciona con ***criterion.*** Consideramos dos posibilidades: Gini y entropy.
- Por último, ***max_features*** nos indica el número de variables a considerar cuando elegimos un separador. Consideramos None (sin restricción),'sqrt' y 'log2', estando este valor en función del número de variables total.

```
param_grid = {'n_estimators': [50, 100, 150],
              'max_depth'   : [None, 3, 10, 20],
              'criterion'   : ['gini', 'entropy'],
              'max_features': ["sqrt","log2",None],
             }
```

Los parámetros óptimos obtenidos serán los siguientes

{'criterion': 'entropy', 'max_depth': None, 'max_features': None, 'n_estimators': 150}

0.8567706755528626 accuracy

Con ello se obtendrá una precisión del 85.68 %. Mostramos ahora la importancia de los predictores.

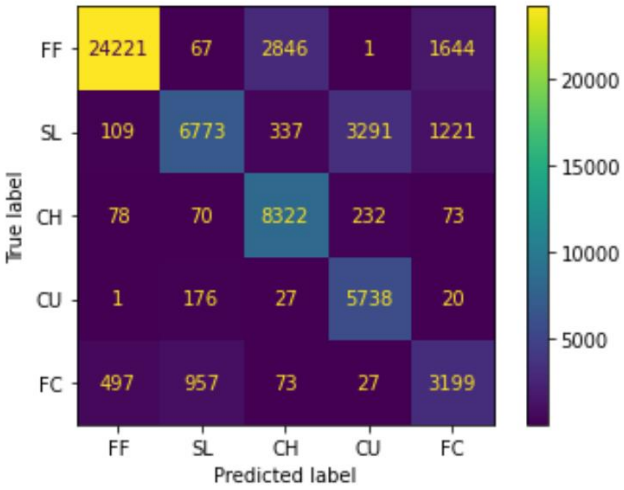
	predictor	importance
5	break_length	0.453496
2	spin_rate	0.095497
14	x0	0.070542
18	pfx_x	0.068660
11	vy0	0.037969
17	z0	0.034792

Es curioso cómo a diferencia del árbol, el primer predictor tendrá una importancia muy superior a los siguientes. Este predictor será la *break_length* con un valor de 0.4535, estando bastante por encima del segundo, el spin_rate con 0.0955

Probaremos a evaluar ahora el caso para el dataset de test, obteniendo una precisión del 80.42 %

Accuracy of RF classifier on test set: 0.8042

La matriz de confusión en el conjunto de test quedará:



Mirando casos concretos podemos ver como para los Fastballs (FF) la precisión baja hasta el 84.2 %, teniendo los mayores errores prediciéndolas como Changeup (CH) y Cutter (FC). Además, el error al clasificar los Sliders se acentúa, llegando a reportarlos erróneamente como Curveballs (CU) en un 28.2 % de las ocasiones. El único tipo de lanzamiento donde parece que el modelo mejora a KNN es el Cutter (FC), ya que la precisión sube al 67.3 %.

Random Forest nos ofrecerá un buen modelo en general pero que parece algo peor que KNN, ya que la precisión general baja entorno a un 1%. Aun así mejora considerablemente a los árboles de decisión y debe ser considerado.

Multilayer Perceptron

El perceptrón multicapa es un tipo de red neuronal que consta de múltiples capas de neuronas interconectadas. Es una de las arquitecturas más comunes y versátiles utilizadas en el campo del aprendizaje profundo. El perceptrón multicapa está compuesto por una capa de entrada, una o más capas ocultas y una capa de salida. La capa de entrada recibe los datos de entrada, mientras que la capa de salida produce las predicciones o resultados deseados.

El entrenamiento del perceptrón multicapa implica ajustar los pesos de las conexiones para minimizar una función de coste o pérdida. Esto se realiza utilizando algoritmos de optimización, como el descenso de gradiente, y técnicas como *backpropagation*, donde el error se propaga hacia atrás en la red para ajustar los pesos en cada capa.

He elegido el perceptrón multicapa ya que ofrece ventajas en términos de su capacidad para aprender relaciones no lineales, además de su flexibilidad y capacidad de generalización. En numerosos problemas, se obtiene una mayor precisión utilizando redes de este tipo, por lo que parece necesario probar su efectividad.

Procedemos primero a elegir los parámetros óptimos para el modelo, que serán los siguientes:

- La distribución de las capas ocultas de la red se especifica en ***hidden_layer_sizes.*** Es complicado, predecir cuántas capas necesitaremos. Generalmente depende de la dificultad del problema a resolver, su linealidad, etc. En este caso he decidido probar 4 casos distintos: 2 con 2 capas ocultas y 2 con 3 capas ocultas, teniendo diferente número de neuronas en cada capa. Así podremos tener una idea de qué será lo más eficiente.
- La función de activación de las neuronas de la capa oculta se elige con ***activation.*** Consideraremos 3 casos distintos, que vimos en la teoría:
  - ***logistic->*** *f(x) = 1 / (1 + exp(-x))*
  - ***tanh->*** *f(x) = tanh(x)*
  - ***relu->****f(x) = max(0, x)*
- El ***solver*** que he decidido usar es ***adam,*** un tipo de optimizador estocástico basado en el gradiente. Según la documentación este ***solver*** funciona relativamente bien con datasets grandes como es este caso, mejorando el tiempo de entrenamiento y la precisión. En los ejemplos de clase utilizábamos ***lbfgs*** porque funciona mejor con datasets pequeños.
- Finalmente queda determinar el factor de aprendizaje con ***learning_rate_init.*** Proponemos valores pequeños como 0.001,0.05 y 0.01.

```
param_grid = {
    'hidden_layer_sizes': [(10,30,10), (40,40), (15,25,15),(20,20)],
    'activation': ['relu','tanh','logistic'],
    'solver': ['adam'],
    'learning_rate_init': [0.001,0.05,0.01],
}
```

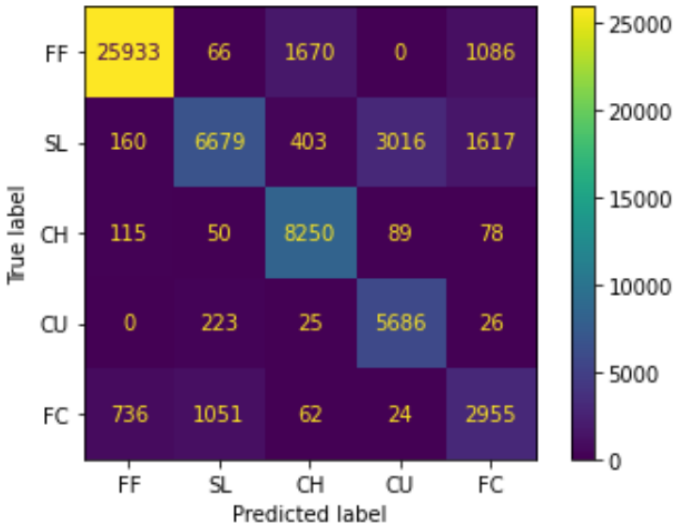
El algoritmo de GridSearch genera los siguientes resultados:

'activation': 'tanh', 'hidden_layer_sizes': (40, 40), 'learning_rate_init': 0.001, 'solver': 'adam'}

Entrenamos el modelo de perceptrón multicapa con estos parámetros y intentamos predecir los valores del conjunto de training y de test.

Accuracy in TRAINING: 0.8517
Accuracy in TEST: 0.8250

Se obtiene una precisión del 85.17 % en el conjunto de entrenamiento y un 82.50 % en el de test. Por otro lado, la matriz de confusión obtenida en el test será la siguiente:



La precisión es mayor que en los algoritmos anteriores, especialmente en el caso de los Fastballs (FF) donde se clasifican de manera correcta un 90.1 % de los casos. Seguimos teniendo problemas aún así con los Sliders, ya que solo conseguimos una efectividad del 56.9 % al clasificarlos.

En general podemos decir que parece ser un modelo algo mejor que los anteriores, logrando una precisión del 82.5 % y que mejora sobre todo la clasificación de las Fastballs (FF).

Comparación de modelos

Ahora que ya hemos entrenado y validado los 4 algoritmos de clasificación, debemos compararlos para tener una idea general de cuál o cuáles de ellos serían adecuados para solucionar el problema en cuestión.

Se muestra una tabla con las precisiones de entrenamiento y de test para los 4 modelos estudiados:

	Precisión Training	Precisión Test
Árbol de Decisión	79.63 %	75.58 %
KNN	83.68 %	81.74 %
Random Forest	85.67 %	80.42 %
Perceptron Multicapa	85.17 %	82.5 %

En principio podemos ver que en todos los modelos no hay una gran diferencia entre los dos tipos de errores, lo que a priori nos indica que hemos conseguido evitar o al menos limitar el sobreaprendizaje de los mismos.

Además, basándonos en el error de Test, el mejor modelo será el del Perceptrón Multicapa con un 82.5 %, seguido de cerca por el de KNN con un 81.74 % y por Random Forest con un 80.42 %. El peor modelo según este criterio será el árbol de decisión, al tener una precisión 5 % inferior.

Volviendo de vuelta al objetivo inicial que fijamos de al menos un 80 % de precisión, podemos ver que tanto KNN como Random Forest y el Perceptron Multicapa cumplen con nuestro objetivo, mientras que el modelo de árboles de decisión se presenta insuficiente.

La elección del modelo adecuado también debe tener en cuenta la interpretabilidad del algoritmo. Por ejemplo, el Perceptron Multicapa quizás es complicado de interpretar, al componerse de numerosos pesos y conexiones complejas. Por otro lado, el árbol de decisión parece el modelo más intuitivo de los 4, al poder obtener una predicción, simplemente respondiendo preguntas sobre las variables.

Conclusiones y Recomendaciones

Para finalizar el proyecto, podemos destacar las siguientes conclusiones y recomendaciones:

- Los árboles de decisión son algoritmos sencillos e intuitivos que serán útiles para problemas simples linealmente separables. Sin embargo, en proyectos como este resultarán en precisiones insuficientes y no deberían usarse por si solos, sino en algoritmos de *Ensemble* como puede ser Random Forest.
- Se consigue alcanzar el objetivo de la precisión mínima en 3 de los 4 algoritmos estudiados: KNN, Random Forest y Perceptrón Multicapa. Siguiendo este criterio cualquiera de los 3 podría ser utilizado con éxito. Aun así, debemos tener cuidado ya que estas precisiones siguen estando cercanas al 80 % y puede que con otro dataset de test nos quedemos por debajo del requerimiento mínimo. Recomendaría intentar buscar unas tasas de acierto algo superiores para tener un margen mayor.
- El Perceptrón Multicapa tiene el mayor acierto de los 4, así que en principio sería el más indicado para usar en este problema. Sin embargo, debemos cuestionarnos también en parte hasta que punto necesitamos interpretabilidad en el modelo, ya que MLP no es capaz de aportarlo. Quizás nos pueda interesar utilizar Random Forest donde perdemos algo de precisión, pero sabemos la importancia de los predictores del modelo entrenado.
- Si esta clasificación de pitches fuese implementada en un entorno profesional, se recomendaría usar los cerca de 3 millones de datos disponibles para intentar minimizar el error en la medida de lo posible. Sin embargo, en este caso no sería apropiado ejecutar los algoritmos en local como estoy haciendo yo. Lo más acertado sería utilizar un clúster (por ejemplo, en la nube) para tener la capacidad computacional necesaria para hacer el procesamiento posible y sin necesitar un tiempo de ejecución desproporcionado.