Manual Técnico del Sistema de Predicción de Apuestas NBA

1. Conceptos clave utilizados en el modelo

En este sistema de predicción de apuestas para la NBA se combinan técnicas de *machine learning* y simulación para estimar las probabilidades de éxito de distintas apuestas. A continuación se explican detalladamente los conceptos fundamentales empleados:

XGBoost (Extreme Gradient Boosting)

XGBoost es un algoritmo de aprendizaje automático basado en potenciación de gradiente que utiliza conjuntos de árboles de decisión para lograr predicciones precisas de manera eficiente [1]. En este sistema se usa **XGBoost** para construir modelos de regresión que predicen estadísticas individuales de los jugadores (puntos, rebotes, asistencias) en un partido dado. Concretamente, se entrena un modelo *XGBRegressor* separado para cada tipo de estadística (PTS, REB, AST), utilizando como *features* diversos datos históricos y contextuales del jugador y el partido [2][3]. XGBoost es útil aquí por su capacidad para manejar muchos *features* y capturar relaciones no lineales, obteniendo modelos robustos que suelen rendir bien en datos tabulares. Gracias a la regularización y al enfoque en reducir el error gradualmente árbol tras árbol, XGBoost tiende a generalizar mejor y evitar *overfitting*, lo cual es crucial en predicciones deportivas donde los datos pueden ser ruidosos.

¿Cómo funciona XGBoost en este contexto? El modelo entrena con datos históricos de partidos (ver sección de *feature engineering*) para aprender la relación entre las características del contexto (ej. promedio de puntos reciente, ritmo de juego del equipo rival, etc.) y el resultado estadístico del jugador. Durante la predicción, dado un nuevo escenario (jugador, oponente, minutos estimados, etc.), cada modelo XGBoost produce un valor esperado para la estadística del jugador. Por ejemplo, el modelo de puntos puede predecir que cierto jugador anotará **~26 puntos** en el próximo partido dado el contexto proporcionado.

Simulaciones Monte Carlo

La simulación de Monte Carlo es un método que utiliza muestreo aleatorio repetido para estimar resultados probabilísticos. En este sistema se emplea una simulación Monte Carlo para estimar la **probabilidad de que el jugador supere una cierta línea de apuesta** (por ejemplo, que haga más de 20 puntos) dado el pronóstico promedio del modelo.

¿Cómo se realiza? El sistema asume que la distribución de las estadísticas del jugador en un partido sigue aproximadamente una **distribución log-normal** (ya que estas métricas son positivas y pueden ser asimétricas). A partir de la predicción del modelo (media esperada) y de una medida de volatilidad (dispersión) para el jugador, se calculan los parámetros de una log-normal equivalente. En código, se obtiene la desviación estándar ajustada `sigma_log` y la media en logaritmo `mu_log` mediante fórmulas de conversión (e.g. `sigma_log = sqrt(ln(1 + (sigma^2/mu^2)))`) [4]. Luego se generan miles de simulaciones aleatorias de esa distribución (por defecto 10.000 muestras) [5]. Cada simulación representa un posible desempeño del jugador (puntos, rebotes o asistencias) en ese partido.

Con los resultados simulados, el sistema calcula la **probabilidad de superar la línea** contando el porcentaje de simulaciones donde la estadística simulada excede la línea de apuesta dada [6]. Además, se extraen percentiles (5º, 25º, mediana, 75º, 95º) para dar una idea de la distribución de resultados posibles [5]. Por ejemplo, si el modelo predice 26 puntos y la volatilidad es moderada, Monte Carlo podría indicar que en un 60% de los escenarios simulados el jugador supera una línea de 25.5 puntos.

Esta técnica es útil porque permite incorporar la incertidumbre y variabilidad del desempeño del jugador en la estimación de probabilidades. En lugar de asumir una distribución fija o un comportamiento determinista, la simulación Monte Carlo genera muchos **escenarios posibles** y estima la probabilidad de éxito de la apuesta de manera numérica. En el código, la función `MonteCarloSimulator.simulate()` retorna justamente esa probabilidad junto con estadísticas de la distribución [7]. Cabe destacar que se ajusta dinámicamente el número de simulaciones según la volatilidad relativa (más simulaciones si el jugador es muy variable) [8], y se limita la desviación para evitar distribuciones excesivamente amplias (máximo 75% de la media) [9], lo cual añade estabilidad a la estimación.

Volatilidad

La **volatilidad** en este contexto se refiere a qué tan variable es el desempeño de un jugador de un partido a otro. Un jugador "volátil" tendrá altibajos marcados (por ejemplo, un día anota 10 puntos y otro 30), mientras que un jugador más consistente tendrá sus cifras más agrupadas cerca del promedio. Cuantificar la volatilidad es importante porque influye en la confianza de nuestras predicciones y en la dispersión de la distribución usada en Monte Carlo.

En el sistema, la volatilidad se calcula como el **coeficiente de variación** (desviación estándar / media) de las estadísticas del jugador [10]. Para lograr una medic obusta, el código combina datos de tres horizontes temporales: - **Volatilidad en la temporada actual:** calculada con los partidos de la temporada en curso (24/25). - **Volatilidad en la temporada anterior:** calculada con los datos de la temporada pasada (23/24). - **Volatilidad en la carrera (histórico total):** calculada con todos los partidos disponibles del jugador.

Cada una de estas se calcula si hay suficientes datos (al menos 5 partidos en cada período) usando la fórmula mencionada (std/mean) [11]. Luego se toma un prom io ponderado de las tres volatilidades: por defecto el sistema asigna hasta un 50% de peso a la volatilidad actual (dependiendo de cuántos juegos de la temporada actual tiene, con un máximo de 40 juegos para peso completo), hasta 30% a la anterior (máximo ~60 juegos para peso completo) y ~20% al histórico total [12]. Estos pesos se normalizan [13] y se calculan los promedios correspondientes [14]. Así se obtiene una **volatilidad combinada** más fiable.

Si un jugador tiene muy pocos partidos (menos de 10 en total), el sistema recurre a valores por defecto de volatilidad (predefinidos en `DEFAULT_VALUES`, por ejemplo 0.25, o 25%, para cada estadística) [15]

De igual forma, si alguna de las volatilidades parciales no se puede calcular por falta de datos, se sustituye por la disponible o un valor por defecto [17]. Finalmente, se realiza un ajuste: se incrementa ligeramente la volatilidad si el jugador tiene muy pocos partidos (multiplicando por un factor hasta 1.2 cuando el total de juegos es bajo) para reflejar la incertidumbre adicional [18]. También se **acota la volatilidad resultante entre 0.15 y 0.45** e 15% a 45%) [19], evitando valores excesivamente bajos (que indicarían certeza irreal) o demas o altos (que darían distribuciones demasiado amplias).

En la práctica, la volatilidad obtenida se usa para determinar la desviación estándar de la distribución log-normal en Monte Carlo. Por ejemplo, si para puntos la volatilidad final es 0.30 (30%) y el modelo predice 20 puntos, la simulación usará una desviación estándar de *sigma = 0.30 * 20 = 6* puntos. Un valor de volatilidad alto implica menor confianza en el promedio pronosticado y por tanto distribuciones más anchas (más probabilidad tanto de superar ampliamente la línea como de quedar muy por debajo), mientras que una volatilidad baja indica que el jugador suele rendir muy cercano a su media esperada.

Ingeniería de características (*Feature Engineering*)

La etapa de *feature engineering* consiste en extraer y crear las **características (features)** que alimentarán al modelo XGBoost. Este sistema construye un conjunto rico de features que combinan estadísticas recientes del jugador, atributos del equipo rival, contexto del partido y otros factores. A continuación, se describen las principales características generadas y su motivación:

- **Promedios móviles del jugador:** Para cada estadística principal (puntos, rebotes, asistencias, y porcentaje de tiro de campo) se calculan promedios históricos:
- *Últimos 5 contra el mismo rival:* promedio en los últimos 5 enfrentamientos del jugador contra ese equipo (si existen) [20]. Esto capt si el jugador tiende a rendir particularmente bien o mal contra ese oponente específico. Por ejemplo, si un jugador suele anotarle muchos puntos a cierto equipo debido a emparejamientos favorables, ese efecto quedará reflejado.
- *Últimos 10 partidos globales:* promedio de las últimas 10 actuaciones del jugador contra cualquier oponente [20]. Represent forma reciente del jugador (rachas de buen rendimiento o slumps).
- *Promedio de la última temporada:* promedio acumulado de la temporada anterior del jugador [21]. Sirve com indicador de su nivel típico histórico, para comparar con su forma actual (por ejemplo, si esta temporada está por encima o debajo de su nivel anterior).

Estos promedios móviles se calculan agrupando por jugador (y rival, cuando aplica) y usando funciones de *rolling mean* o *expanding mean* con desplazamiento de 1 partido para no incluir el juego actual [22].
Las columnas resultantes reciben nombres como `pts_last5_vs_opp`, `reb_last10`, `ast_last_season`, etc. Cualquier valor faltante (por ejemplo, si el jugador no tiene 5 enfrentamientos previos contra ese rival) se rellena con un valor por defecto razonable (e.g. 0 o un promedio base) [23] para evitar nulos.

- **Días de descanso:** Se añaden dos features que indican el descanso en días tanto del jugador como de su equipo rival antes del partido:
- `dias_descanso_jugador`: días desde el último partido jugado por ese jugador [24].
- `dias_descanso_rival`: días desde el último partido del equipo rival [25].

Estas características reflejan el efecto de la fatiga o frescura. Un jugador que viene de jugar ayer (0 días de descanso) podría rendir peor por cansancio, mientras que con varios días de descanso podría rendir mejor. De igual forma, si el equipo rival jugó recientemente, podría defender con menos eficacia. En el código se calculan ordenando por fechas y restando fechas consecutivas por jugador y por equipo, acotando luego entre 0 y 10 días [26] (valores mayores a 10 se dejan en 10, asumiendo que más de 10 días y aportan mejora adicional).

- **Datos del jugador (perfil):** Del archivo `jugadores.txt` se cargan estadísticas avanzadas promedio de cada jugador, tales como:
- **USG% (Uso ofensivo):** porcentaje de posesiones del equipo que el jugador usa (tiros o pérdidas) cuando está en cancha. Un USG% alto indica que el jugador tiene un rol protagónico en la ofensiva, lo cual suele correlacionar con mayores estadísticas contables (puntos especialmente).
- **TS% (True Shooting %):** medida de eficiencia de tiro del jugador considerando tiros de 2, 3 y libres. Un TS% alto indica que anota eficientemente, útil para contextualizar si sus promedios de puntos vienen con buena efectividad.
- **AST% y REB%:** porcentaje de asistencias y rebotes que aporta cuando está en cancha. Por ejemplo, AST% representa la proporción de canastas de sus compañeros que asiste, lo que indica su rol como generador de juego; REB% indica qué porcentaje de rebotes totales captura, reflejando su presencia en los tableros.

Estos datos funcionan como *features estáticas* que caracterizan el estilo del jugador. Se integran al conjunto de características durante el merge de datos (añadidos desde `jugadores.txt` por nombre del jugador) [27] [28]. En el conjunto final, a en como columnas `USG_perc`, `TS_perc_x` (el sufijo `_x` diferencia que vienen del DF de jugadores), `AST_perc_x`, `REB_perc_x`, etc.

- **Datos del equipo rival:** Del archivo `equipos.txt` se incorporan métricas del equipo contrario que pueden afectar el desempeño del jugador:

- **OffRtg y DefRtg:** Calificaciones ofensiva y defensiva del equipo rival (puntos anotados o permitidos por 100 posesiones). Un rival con DefRtg alto (defensa débil) típicamente permite más facilidades, lo que podría elevar las estadísticas del jugador.
- **PACE (Ritmo de juego):** número de posesiones promedio por partido del equipo. Un Pace alto implica más posesiones en el partido, y por tanto más oportunidades de acumular puntos, rebotes, asistencias.
- **AST_perc, AST_TO:** Porcentaje de asistencias y ratio asistencia/pérdida del equipo. Aunque son datos más globales, podrían indicar estilo de juego (equipos que permiten más uno contra uno vs. equipos que fuerzan más asistencias).
- **REB_perc:** Porcentaje de rebotes disponibles que captura el equipo, lo cual da una idea de si el rival domina o cede rebotes (afectando las oportunidades de rebote del jugador en cuestión).
- **eFG% y TS% del equipo:** la eficacia de tiro del equipo rival, que puede influir en el número de rebotes disponibles (un equipo rival que tira muy bien deja menos rebotes defensivos para capturar, por ejemplo).
- **PIE (Player Impact Estimate) del equipo:** una métrica global del impacto del equipo en el juego (similar a porcentaje de la estadística total del partido), que podría correlacionar con qué tanto dominan los partidos.

Estos datos de equipo se añaden emparejando el `oponente` del partido con la fila correspondiente en `equipos.txt` [27]. En las columnas finales aparecen con sufijo `_y` para distinguirlos (ej. `AST_perc_y` para AST% del equipo, `REB_perc_y` , etc.) [28]. La idea es capturar la **fortaleza o debilidades del rival**: por ejemplo, contra un equipo de ritmo alto y mala defensa, podemos esperar que el jugador exceda un poco sus promedios.

- **Posición vs oponente:** Del archivo `posiciones.txt` se agrega, en función de la posición del jugador y el equipo rival, el promedio de puntos, rebotes y asistencias que ese equipo suele permitir a jugadores de esa posición. Por ejemplo, `ptspp` (puntos por partido) podría indicar "el equipo X permite 20.1 puntos en promedio a escoltas rivales". Las columnas `ptspp` , `rebpp` , `asistpp` se obtienen buscando la fila que coincide con el equipo oponente y la posición del jugador en cuestión [29] [30]. Esto es muy útil para ajustar las expectativas: si un pívot se enfrenta a un equipo contra el cual los pívots suelen hacer muchos puntos y rebotes, el modelo puede aumentar su predicción acorde a ello. Estas variables introducen conocimiento específico del **emparejamiento posicional**.

- **Contexto del partido:** Otras variables contextuales incluyen:

- **Localía (`local`):** indica con 1/0 si el jugador juega en casa. Los jugadores a veces rinden mejor de local (por comodidad, apoyo del público) o peor de visitante. El modelo considerará esta variable si en los datos históricos se perciben diferencias de rendimiento entre casa y fuera.
- **Último resultado del equipo (`ultimo_resultado_equipo`):** en el código está inicializado como 0 para todos los casos [31], pues no se disponía de la secuencia de victorias/derrotas. Sin embargo, la idea es que podría incorporar si el equipo del jugador viene de ganar (1) o perder (0) el partido anterior, como indicador de moral o tendencia. Actualmente no se utiliza efectivamente (queda como campo constante 0), pero está preparado para futuras mejoras con datos de racha del equipo.
- **Partidos jugados (`partidos_temporada` y `partidos_totales`):** representan el número de partidos del jugador en la temporada en curso y en total en el dataset [32]. Esto puede ayudar a diferenciar jugadores consolidados de aquellos con menos experiencia o menos datos (por ejemplo, un rookie con pocos partidos puede tener predicciones menos confiables). También permite al modelo ajustar expectativas en función de la carga de juego (un jugador con muchos partidos puede estar más cansado, aunque eso no está explícitamente codificado, el modelo podría captarlo indirectamente).

En total, tras la ingeniería de características, se construye un *vector de features* muy descriptivo para cada instancia jugador-partido. El sistema asegura que solo se utilizan las columnas relevantes (por ejemplo, si alguna columna faltara por datos inexistentes, se excluye) [33] [34]. Gracias a este amplio conjunto de *features*, el modelo XGBoost puede aprender

patrones complejos: rachas de rendimiento, influencias del rival, efectos de descanso, etc. Esto mejora la precisión de las predicciones en comparación con usar solo simples promedios históricos.

2. Análisis del funcionamiento de `cuotas.py`

El archivo **`cuotas.py`** implementa un subsistema de aprendizaje que **retroalimenta el modelo** con el historial de apuestas realizadas, utilizando un *XGBClassifier* para mejorar la estimación de probabilidades de acierto de futuras apuestas. En esencia, mientras `predictor.py` genera la predicción inicial (probabilidad de over, EV, etc.), `cuotas.py` aprende de los **aciertos y fallos pasados** para ajustar o complementar esas predicciones con un modelo de clasificación binaria (apuesta ganada o perdida).

Aprendizaje de histórico de apuestas: El sistema lleva un registro de cada apuesta en un archivo CSV (`apuestas_log_simple.csv`) donde se anotan: jugador, rival, tipo de apuesta (stat), línea, cuota ofrecida, probabilidad estimada por el modelo, valor esperado (EV), resultado (acierto 1/0 una vez sabido) y otros datos. `cuotas.py` carga este log y, cuando hay suficientes apuestas registradas (por diseño al menos 100 casos mínimos para empezar a entrenar) [35] , construye un conjunto de *features* específico p alimentar un modelo **XGBClassifier** que intentará predecir la columna *acierto* (si la apuesta resulta ganadora o no).

Las principales etapas y características de `cuotas.py` son:

- **Preparación de datos:** La función `preparar_features_apuestas(df)` toma el DataFrame del log de apuestas y calcula varias características derivadas:
- **Cuota ofrecida y redondeada:** se incluye directamente la `cuota_ofrecida` de la apuesta y también una versión redondeada a 2 decimales (`cuota_round`) [36] . Esto permite agrupar apuestas con cuotas similares.
- **Probabilidad y EV estimados:** se llevan como features la `probabilidad` estimada por el modelo principal y el `valor_esperado_EV` correspondiente [37] , para que el clasificador sepa qué predijo el modelo base.
- **Acierto histórico por cuota (`acierto_cuota`):** se agrupan todas las apuestas del log por cuota (redondeada) y se calcula el porcentaje de acierto histórico de cada nivel de cuota [38] . Por ejemplo, si to s las apuestas con cuota ~1.80 han resultado ganadoras el 55% de las veces, `acierto_cuota` para una apuesta de cuota 1.80 será 0.55. Esto ayuda a calibrar si ciertas cuotas tienden a estar infravaloradas o sobrevaloradas en nuestro histórico.
- **Racha del jugador-stat:** se calculan métricas de *hit rate* (porcentaje de acierto) recientes para el mismo jugador y estadística:
  `hit_jugador_last10` : fracción de aciertos del mismo jugador en esa estadística en sus últimas 10 apuestas registradas [39] . `hi _jugador_last5` : lo mismo pero en las últimas 5 apuestas [40] .
  Estos valores dan idea de si últimamente las predicciones para ese jugador han sido acertadas o no, quizá reflejando tendencias no capturadas por el modelo principal (por ejemplo, un jugador que el modelo sigue pronosticando alto pero lleva varios unders seguidos podría tener un hit rate bajo reciente).
- **Historial vs rival:** `hit_jugador_vs_rival` : similar a lo anterior pero filtra apuestas del mismo jugador y estadística **contra el mismo rival**, calculando el porcentaje de acierto en hasta 5 apuestas previas vs ese equipo [41] . Esto puede capturar patrones específicos de ese enfrentamiento (ej: siempre fallamos al predecir over de rebotes de cierto jugador contra cierto equipo).
- **Hit global por tipo de stat:** `hit_global_stat` calcula la tasa de acierto global de todas las apuestas de ese tipo (pts, reb o ast) en una ventana de las últimas 20 apuestas de ese stat [42] . Indica si en general nuestras apuestas de, por ejemplo, puntos están siendo exitosas recientemente. Si este valor es bajo, podría sugerir un sesgo del modelo en esa estadística.

- **Probabilidad XGB previa:** `proba_XGB_prev` es un feature que incorpora la predicción anterior del mismo clasificador para apuestas del mismo jugador y stat [43]. Dado que cada vez que corremos `cuotas.py` obtenemos una probabilidad *proba_XGB* para cada apuesta, al tomar el histórico podemos desplazar esa columna una posición hacia abajo por jugador+stat, de modo que cada fila tenga la *proba* que el clasificador dio en la última apuesta similar anterior [43]. Si no existe un valor previo (primera vez que apostamos a ese jugador-stat), se pone -1. Esto permite al modelo saber si en la ocasión anterior ya pronosticaba algo (por ejemplo, quizá predijo 0.8 de probabilidad de acierto y aún así falló, etc.).

Todos estos features se rellenan con -1 donde no haya datos suficientes (por ejemplo, si un jugador solo tiene 2 apuestas previas, el *rolling* a 5 o 10 tendrá NaNs, que se reemplazan por -1 para indicar "sin historial") [44]. Finalmente se selecciona un subconjunto de columnas relevantes para entrenar el modelo: cuota, probabilidad, EV, los hit rates calculados, acierto_cuota y proba_XGB_prev [45].

- **Entrenamiento del XGBClassifier:** Una vez construidos los features, el flujo en `cuotas.py` es:
- **Condición de mínimo de datos:** si el log tiene menos de *N_MINIMO* apuestas registradas (100 por defecto), no se entrena nada por falta de datos suficientes [35]. En ese caso simplemente se añade una columna vacía o cero de `proba_XGB` al CSV.
- Si hay datos suficientes, prepara el *dataset* de entrenamiento. Importante: se evita usar las primeras 100 apuestas para entrenar (quizás como período de calentamiento). El código toma las filas desde el índice 100 en adelante (`loc[N_MINIMO:]`) como válidas para entrenamiento. Esto significa que los primeros 100 registros se dejan para inicializar features como *rolling* sin influencia en el modelo, y el modelo empieza a aprender a partir de la apuesta 101.
- Se entrena el clasificador XGBoost con estos datos: utilizando 100 árboles, profundidad 3, learning rate 0.12, etc. (parámetros predefinidos) [47] para predecir la columna *acierto*. Durante el *fit*, el modelo aprende qué combinación de features tiende a indicar una apuesta ganada.
- **Predicción con el clasificador:** Luego de entrenado, se usa para predecir la probabilidad de acierto de **todas las apuestas a partir de la número 100 en adelante** (incluyendo las más recientes aún sin saber el resultado) [48]. En concreto, calcula `model.predict_proba` para obtener la probabilidad estimada de clase 1 (acierto) y la agrega al dataframe del log en la columna `proba_XGB` [48]. Las apuestas anteriores al índice 100 se les asigna `proba_XGB = 0` simplemente para completar la columna sin entrenar con ellas [48].
- Se sobreescribe el archivo CSV de log con la nueva columna `proba_XGB` actualizada para cada apuesta [49].
- **Salida e interpretación:** Al ejecutar `cuotas.py` , tras entrenar y actualizar el log, el script imprime la probabilidad XGB calculada **para las apuestas del día actual** [50]. Es decir, si agregamos nuevas apuestas con la fecha de hoy (o se registraron hoy al correr el predictor), mostrará en consola algo como un listado de probabilidades para cada apuesta de hoy. Por ejemplo:

```
proba_XGB
120    0.612
121    0.455
```

Esto indicaría que, según el clasificador entrenado con todo el histórico, la apuesta con índice 120 tiene ~61.2% de probabilidad de ser ganadora, y la 121 un ~45.5%. Esta *proba_XGB* tiene en cuenta factores como los patrones de aciertos previos, cuotas, etc., por lo que sirve como **segunda opinión** o ajuste a la probabilidad original del modelo.

En resumen, `cuotas.py` implementa un **modelo de calibración y aprendizaje continuo**: con cada nueva apuesta cuyos resultados se van añadiendo, el XGBClassifier refina su capacidad de detectar qué señales llevan a un acierto. Por ejemplo,

podría aprender si cierto jugador tiende a no alcanzar sus overs a pesar de probabilidades altas, o si cuando el EV es muy alto suele ser un indicio de valor real, etc. A medida que se acumule historial (p.ej. cientos de apuestas), este modelo podría ayudar a ajustar las decisiones del sistema (por ahora, se registra y se muestra, pero potencialmente se podría usar *proba_XGB* para modular las recomendaciones de apostar o no apostar). En la práctica actual, el resultado de `cuotas.py` le sirve al usuario para revisar la confianza ajustada en sus apuestas más recientes basado en la experiencia acumulada.

Además, el *log* de apuestas puede analizarse con la función `analizar_log_apuestas()` (llamando la opción 2 en el menú del programa principal). Esta función calcula estadísticas globales: número de apuestas, porcentaje de aciertos global, EV medio de las apuestas y beneficio acumulado histórico [51]

Esto permite llevar un control de rendimiento general del sistema (por ejemplo, *"llevamos un 55% de aciertos globales y un EV medio de 0.05 (+5%) por apuesta"*).

3. Descripción técnica de `predictor.py` y estructura del sistema

El archivo **`predictor.py`** contiene la implementación principal del sistema de predicción. Está organizado en clases que se encargan de diferentes responsabilidades (cargar datos, generar features, entrenar modelos, hacer predicciones y simular probabilidades). A continuación describimos las clases clave y cómo interactúan entre sí para producir las predicciones:

- **NBADataLoader:** Esta clase gestiona la carga y preparación de los datos base desde los archivos CSV/TXT:
- Su método `load_all_data()` lee los cuatro archivos principales de datos (`partidos.csv` , `jugadores.txt` , `equipos.txt` , `posiciones.txt`) [53], utilizando métodos internos separados para cada uno. Durante la carga:

  Renombra columnas del archivo de partidos a nombres consistentes (por ejemplo, "Player" a "jugador", "Opp" a "oponente", "fgp" a "FG_perc", etc.) [54] [55], convierte tipos de datos (numéricos y fechas [56], limpia texto (nombres en mayúsculas sin espacios extra) [57], y descarta filas incompletas [58].
  Lee `jugadores.txt` (sin cabecera, columnas predefinidas) y asigna nombres a columnas según `COLS_JUGADORES` [59]. Luego convierte las columnas numéricas (USG%, TS%, etc.) a tipo numérico y descarta filas inválidas [60].
  Lee `equipos.txt` de forma similar, asignando `COLS_EQUIPOS` [61]. Si existe una columna "POSS" (posesiones totales) la elimina, ya que no se usa [62]. Convierte columnas numéricas (Ofensiva, Defensiva, Ritmo, etc.) y limpia textos (códigos de equipo) [63].
  Lee `posiciones.txt` con columnas `COLS_POSICIONES` [64], limpia texto (TEAM, pos) y pasa a numérico los promedios de pts, reb, asist por posición.
  Valida que todos los DataFrames tengan las columnas requeridas [65] [66], lanzando error si falta algo importante.
  Finalmente, llama a `add_descanso_features()` para calcular los días de descanso del jugador y rival directamente en el DataFrame de partidos cargado [67] (esta es la misma lógica descrita en *feature engineering*, aplicada en la carga para que estos datos queden disponibles).

- El resultado de `load_all_data()` es la tupla de DataFrames `(df_partidos, df_jugadores, df_equipos, df_posiciones)` lista para usar [68]. En la ejecución principal, se instancia `NBADataLoader` y se obtienen estos DataFrames para procesarlos.

- **FeatureEngineer:** Esta clase toma los DataFrames cargados y construye el *feature set* final para el modelo:

- En su inicialización, crea un `LabelEncoder` para los equipos (oponentes) y lo ajusta con todos los nombres de equipo presentes tanto en partidos como en el DF de equipos [69]. Esto permite convertir el equipo rival a un **número entero identificador** (`oponente_enc`) en lugar de cadena, lo que es necesario para usarlo como feature en el modelo [33].

- Sus métodos principales ya se discutieron en la sección de *feature engineering*: `add_rolling_features()` agrega los promedios móviles (last5_vs_opp, last10, last_season) [22], `add_descanso_features()` agrega los días de descanso [24], y luego `construir_features()` combina todos los datos:

    1. Llama a los dos métodos anteriores para que `self.df` (copia de df_partidos) ya tenga las columnas de histórico y descanso [70].
    2. Hace *merge* de `self.df` con `df_jugadores` (por nombre de jugador), con `df_equipos` (uniendo oponente con TEAM) y con `df_posiciones` (uniendo oponente & pos jugador) [27]. El resultado `merged` contiene en una sola tabla todos los datos relevantes del partido, el jugador y el contexto.
    3. Añade las columnas calculadas adicionales como `ultimo_resultado_equipo` , `partidos_temporada` y `partidos_totales` [32].
    4. Selecciona únicamente las columnas de features que se van a usar en el modelo, ignorando las demás. Este paso utiliza la lista predefinida de features relevantes (minutos, encoding de oponente, stats de jugador, stats de equipo, features de histórico, etc.) [28] y filtra el DataFrame combinado [33] [34].
    5. Guarda el DataFrame resultante en `self.features` y lo retorna.

- Además, `FeatureEngineer` tiene `get_targets()` , que simplemente devuelve un diccionario con las columnas objetivo reales de puntos, rebotes y asistencias del DataFrame de partidos original [71]. Es decir, prepara `y_dict = {'pts': df['pts'], 'reb': df['reb'], 'ast': df['ast']}` como las series de valores que el modelo debe aprender a predecir.

- **NBAModelTrainer:** Esta clase entrena los modelos XGBoost de predicción de estadísticas:

- Toma como entrada el DataFrame de features `X` y el diccionario de targets `y_dict` . Tiene la capacidad de filtrar los datos (parámetro opcional `df_filtrado`) si se quisiera entrenar con un subconjunto (en este caso no se usa, entrena con todo el historial disponible) [72].

- Su método `train_models()` realiza la división entrenamiento/test y entrena tres modelos:

    Realiza un `train_test_split` para cada estadística por separado (usando 80% datos para entrenar, 20% para prueba, semilla fija para reproducibilidad) [73].
    Crea y entrena un modelo XGBRegressor para puntos, otro para rebotes y otro para asistencias con los mismos parámetros predeterminados (`DEFAULT_MODEL_PARAMS`) definidos al inicio, por ejemplo 150 árboles, profundidad 3, learning rate 0.1, etc. [74] [75].
    Cada modelo se entrena para ajustar las predicciones a su respectiva variable objetivo. Al terminar, guarda los modelos entrenados en un diccionario (`self.models`) y opcionalmente exporta cada modelo a un archivo JSON en disco (directorio `models/`)
    Esto permite reutilizar modelos ya entrenados sin reentrenar cada vez, si así se deseara.
    También hace logging de métricas de desempeño (R^2 en el conjunto de prueba para cada modelo) [74].
    En la versión actual, las métricas se imprimen via logging, y existe un método `evaluate_models()` para imprimir MAE, MSE y las 5 features más importantes de cada modelo [77] [78], útil para diagnosticar el comportamiento del modelo y ver qué factores está considerando más.

- El resultado de `train_models()` es el diccionario `models` con tres modelos XGBoost entrenados (asociados a 'pts', 'reb', 'ast'), listos para hacer predicciones.

- **NBAPredictor:** Esta clase integra los modelos entrenados y el *FeatureEngineer* para realizar la **predicción para un nuevo partido/apuesta**. Cuando el usuario ingresa un jugador, rival y contexto del partido, `NBAPredictor` se encarga de:

- En su construcción (`__init__`), simplemente almacena los modelos y referencia al objeto FeatureEngineer (y guarda también `df` original para consultas) [79].

- El método principal es `predict(input_data)` que devuelve dos cosas:

  1. `preds` : un diccionario con las predicciones numéricas de estadísticas (puntos, rebotes, asistencias) para el jugador en ese partido.
  2. `vol` : un diccionario con las volatilidades calculadas para esas estadísticas.

  Internamente este método: - Prepara un DataFrame de *features* para el partido de interés mediante `_prepare_features(input_data)` [80]. - Garantiza que el orden de columnas en este DataFrame coincida con el que el modelo espera (misma secuencia que `feature_engineer.features` original) [80], ya que XGBoost es sensible al orden de columnas. - Llama a `_make_predictions(features)` que aplica cada modelo XGBoost para obtener los valores pronosticados de pts, reb y ast (tomando el primer elemento de la predicción ya que es un DataFrame de una fila) [81]. Se aplica `max(0, prediccion)` para no dar valores negativos por seguridad [81]. - Llama a `_calculate_volatility(jugador)` para obtener las volatilidades de ese jugador [80], usando el método descrito anteriormente. - Finalmente retorna ambos resultados.

- `_prepare_features(input_data)` : este es un paso crítico donde, dado el input del usuario (que incluye `'jugador'` (nombre), `'rival'` (equipo rival), `'mins'` (minutos estimados que jugará), `'local'` (1 o 0 indicando si juega en casa) y `'ult_resultado'` (1 victoria previa del equipo, 0 derrota previa)), construye la fila de features correspondiente:

  Normaliza el nombre del jugador y equipo a mayúsculas para hacer matching [82]. Busca en `df_jugadores` los stats avanzados del jugador; si no se encuentra (jugador nuevo no en la lista), utiliza valores por defecto neutros [83]. Esto llena `stats_j` con U%, TS%, etc. del jugador.
  Busca en `df_equipos` los stats del equipo rival; si el código de equipo no se encuentra (por ejemplo, equipo nuevo o error), pone ceros por defecto [84]. Llena `stats_e` con OffRtg, DefRtg, Pace, etc. del equipo rival.
  Determina la posición del jugador y busca en `df_posiciones` los promedios del rival contra esa posición (`stats_pos`); si no encuentra (por ejemplo, posición no estándar o equipo sin datos), usa los valores default de pts, reb, ast medios [85].
  Recupera datos de los últimos partidos del jugador mediante `_get_last_game_data(jugador, rival)` [86]. Esta función consulta en el DataFrame histórico:
  Toma los últimos 10 partidos globales del jugador y calcula sus promedios de pts, reb, ast, FG% [87] [88].
  Toma los últimos 5 partidos del jugador contra ese mismo rival y calcula promedios (si existen) [89].
  Toma promedios de la última temporada completa y de la temporada actual por separado [90] [91].
  Cuenta cuántos partidos jugó en la temporada actual y en total [89] [92].
  Rellena cualquier valor faltante: si no hubo enfrentamientos previos contra el rival actual, por ejemplo, pone un valor por defecto o recicla el promedio global [93].

Retorna un diccionario `last_data` con keys como 'pts_last10', 'reb_last5_vs_opp', 'pts_last_season', etc.

Con toda esta información (`stats_j` , `stats_e` , `stats_pos` , `last_data`), construye un diccionario `features` correspondiente a la nueva instancia:

Asigna el valor de minutos estimados `'min'` del input.

Codifica el equipo rival con el LabelEncoder (`oponente_enc`) [94].

Inserta las estadísticas avanzadas del jugador (USG_perc, TS_perc, AST_perc_x, REB_perc_x) [95]. En el diccionario de features, las claves con `_x` representan stats del jugador; por ejemplo `AST_perc_x` es la tasa de asistencias del jugador. Inserta las estadísticas del equipo rival (OffRtg, DefRtg, PACE, AST_perc_y, AST_TO, REB_perc_y, eFG_perc, PIE) [96] (`_y` indicando stats del equipo).

Inserta las features de histórico obtenidas en `last_data` : pts_last5_vs_opp, reb_last10, ast_last_season, FG_perc_last10, etc. [97] [98]. Para cada una, si en `last_data` faltaba y se puso default, aquí simplemente se toma ese valor.

Inserta los datos de posición vs oponente: `ptspp` , `rebpp` , `asistpp` [99].

Finalmente agrega las variables de contexto del input: `local` (ya viene en input), `dias_descanso_jugador` y `dias_descanso_rival` calculados a partir de los datos históricos (usando `_get_last_rest_jugador` y `_get_last_rest_rival` , que básicamente buscan en el DataFrame la última fila del jugador o del equipo rival y leen los días de descanso almacenados) [100] [101]. También `ultimo_resultado_equipo` tomado del input (indicador de victoria/derrota anterior) [102], y los conteos de partidos de `last_data` (`partidos_temporada` y `partidos_totales`) [103].

Todos estos valores se consolidan en un DataFrame de una fila [104], que representa las *features* del partido a predecir, con exactamente las mismas columnas que el conjunto de entrenamiento tenía. Este es el DataFrame que luego alimenta a los modelos XGBoost para obtener las predicciones.

- `_make_predictions(features)` : como mencionado, aplica cada modelo regressor a la fila de features y devuelve un diccionario `{'pts': pred_pts, 'reb': pred_reb, 'ast': pred_ast}` con valores numéricos (floats) de las predicciones [81]. Por ejemplo, podría devolver `{'pts': 25.8, 'reb': 7.1, 'ast': 5.3}` indicando el rendimiento esperado del jugador.

- `_calculate_volatility(jugador)` : calcula la volatilidad del jugador para cada estadística usando el método antes descrito (combinando temporadas, etc.) [11] [19]. Este método es llamado dentro de `predict()` para obtener la volatilidad *actualizada a ese jugador* cada vez que se va a simular. Notar que usa todos los datos históricos del jugador disponibles (hasta la fecha), por lo que si tras la última actualización de `partidos.csv` el jugador incrementó su variabilidad, quedará reflejado.

- Otros métodos auxiliares: `_get_player_stats` , `_get_team_stats` , `_get_position_stats` ya mencionados (buscan en los DataFrames correspondientes el registro de jugador/equipo/posición) [105] [106] . Si no encuentran un registro (por ejemplo, jugador nuevo sin fila en jugadores.txt), lanzan excepción que es manejada en `_prepare_features` para usar defaults [83]. También `_get_last_game_data` ya descrito, y `_handle_error` simplemente imprime el error si algo falla en predicción [107].

- **MonteCarloSimulator:** Ya detallamos su funcionamiento: genera simulaciones log-normales para cada stat usando la media predicha y la desviación (media * volatilidad) [108], y produce la probabilidad de superar la línea y percentiles [4] . En la integración, tras obtener `preds` y `vol` del predictor, el código principal instancia `MonteCarloSimulator` (semilla 42 para reproducibilidad) y llama a

`calculate_probabilities(preds, vol, lines)` donde `lines` son las líneas de apuesta ingresadas. Esto retorna un diccionario `results` con, para cada stat, la probabilidad de over, percentiles, etc.

Estas clases trabajan en conjunto en el **flujo principal** del programa (cuando el usuario ejecuta la opción de predicción): 1. Se carga y prepara la data (NBADataLoader). 2. Se generan las features y objetivos (FeatureEngineer). 3. Se entrenan o cargan los modelos predictivos (NBAModelTrainer). 4. Se solicita la entrada del usuario (jugador, rival, minutos, etc.) con `get_user_input()` [109] [110]. Este es un simple prompt interactivo en consola que devuelve el diccionario `input_data`. 5. Se crea `NBAPredictor` con los modelos entrenados y el feature engineer ya configurado. 6. Se llama `predictor.predict(input_data)` para obtener las predicciones del jugador y su volatilidad. 7. Se pide al usuario ingresar las **líneas de apuesta y cuotas ofrecidas** para puntos, rebotes, asistencias mediante `get_lines_input()` [111]. Este solicita por consola cada línea (e.g. "Over Puntos línea:") y cuota correspondiente, devolviendo un diccionario `lines` con, por ejemplo, `{'pts': 25.5, 'pts_cuota': 1.85, 'reb': 7.5, 'reb_cuota': 1.90, ...}`. 8. Se utiliza el simulador Monte Carlo: `results = montecarlo.calculate_probabilities(preds, vol, lines)` para calcular la probabilidad de *over* para cada estadística con su distribución. 9. Se carga el log histórico de apuestas con `cargar_log_apuestas()` para tenerlo disponible en la generación de la salida. Esta función intenta leer `apuestas_log_simple.csv` y convertir a numérico ciertas columnas (valor_esperado, cuota_ofrecida, acierto) [112], devolviendo un DataFrame (o vacío si no existe aún, en cuyo caso se avisa que no habrá histórico disponible). 10. Finalmente, se llama a `display_results_con_historial(preds, vol, results, lines, df, jugador, rival, df_log)` que imprime en la consola todos los resultados de la predicción de forma legible [113]. Este paso: - Muestra advertencias si el jugador tiene pocos partidos en el histórico (menos de 10 como advertencia de poca fiabilidad, menos de 20 como información) [114]. - Muestra las estadísticas predichas: *"Puntos: X | Rebotes: Y | Asistencias: Z"* [115]. - Muestra las volatilidades calculadas para cada stat (marcando que >0.40 es alta volatilidad) [116]. - Por cada categoría (PTS, REB, AST), imprime: * La línea de apuesta y la probabilidad estimada de superar esa línea (en porcentaje) [117]. * La **cuota justa** calculada según el modelo, que es simplemente el inverso de la probabilidad (`1/probabilidad`), indicando qué cuota sería "justa" dada dicha probabilidad [118]. Por ejemplo, si la probabilidad de over es 60%, la cuota justa sería ~1.67. * La **cuota ofrecida** ingresada por el usuario y el **Valor Esperado (EV)** de la apuesta, que se calcula como `probabilidad * cuota_ofrecida - 1` [119]. Este EV representa el retorno promedio esperado por cada unidad apostada: un EV positivo (>0) implica que la apuesta tiene valor a largo plazo (probabilidad * cuota > 1), mientras que un EV negativo sugiere que la casa de apuestas tiene la ventaja. * Una categorización de **"seguridad"** de la apuesta en términos cualitativos (Alta, Media, Moderada o Baja) según la probabilidad y volatilidad [120]. Por ejemplo, si la probabilidad estimada es muy alta (>62%) y la volatilidad del jugador es baja (<0.28), se etiqueta como "Alta" seguridad [120]. Esto ayuda a saber qué tan confiable es la predicción: una probabilidad moderada pero con jugador muy volátil reduciría la confianza. * Métricas históricas si existen: utilizando `obtener_metricas_historicas(df_log, stat)` se filtra el histórico por ese tipo de apuesta (stat) y opcionalmente por EV mínimo (en el código usan ev_min=0.0 para considerar todas) [121]. Si hay suficiente datos en histórico, imprime: - Número de apuestas similares registradas y el % de acierto histórico en ellas [122]. - EV medio histórico y beneficio acumulado obtenido en esas apuestas [122]. * En base a lo anterior, el sistema da una **recomendación final**: si el EV actual es positivo y además el historial para ese tipo de apuesta es favorable (>50% acierto y beneficio positivo), recomienda **"APOSTAR"**, resaltado en verde en consola [123]. De lo contrario, recomienda **"NO apostar"** (en rojo) [124]. Esta recomendación combina la expectativa teórica con la evidencia empírica de nuestro propio desempeño. * Muestra también los percentiles simulados (p. ej. *"5%: 15 | 25%: 22 | 50%: 26 | 75%: 30 | 95%: 35"* puntos) para dar una idea de la distribución de posibles resultados [125], y la cantidad de simulaciones realizadas [126]. - Por último, llama a `guardar_apuesta(...)` para **registrar la apuesta evaluada en el log** [127]. Se almacena una nueva fila en `apuestas_log_simple.csv` con fecha y hora actual, jugador, rival, tipo de stat, línea, cuota ofrecida, cuota justa calculada, probabilidad modelo, EV, recomendación dada, seguridad, acierto (se pone 0 por defecto asumiendo pendiente, luego el usuario deberá editarla cuando se sepa el resultado) y proba_XGB vacía [128] [129]. De este modo, cada predicción hecha queda documentada para posteriormente alimentar el análisis histórico y el modelo de `cuotas.py`.

En síntesis, `predictor.py` automatiza todo el proceso desde la lectura de datos hasta la entrega de una **predicción detallada** por pantalla. El diseño modular permite que cada parte (datos, features, modelo, simulación) sea mantenible y reemplazable. Por ejemplo, se podrían actualizar los datos o recalibrar el modelo sin cambiar la lógica de simulación; o ajustar la estrategia de recomendaciones sin tocar la parte de predicción base.

4. Uso de APIs y actualización de datos

Actualmente, el sistema se nutre de archivos estáticos (`partidos.csv` , etc.) que necesitan ser actualizados periódicamente de forma manual. Sin embargo, es **deseable mantener los datos lo más actualizados posible**, especialmente los registros de los últimos partidos, para que las *features* de forma reciente (last5, last10) y descanso reflejen la realidad actual. Si el sistema no se actualiza, las predicciones pueden degradarse con el tiempo (por ejemplo, un jugador puede cambiar de equipo o rol, o un equipo puede mejorar/emp empeorar su defensa a lo largo de la temporada).

Actualización manual: Se recomienda después de cada jornada de la NBA (o al menos cada semana) agregar las nuevas filas de partidos al archivo `partidos.csv` . Este archivo contiene el boxscore de cada partido por jugador, por lo que habría que incorporar la línea estadística de cada jugador relevante de los partidos más recientes (puntos, rebotes, asistencias, etc., con la misma estructura de columnas y formateo de fecha). Igualmente, si hay jugadores nuevos emergiendo, actualizar `jugadores.txt` con sus estadísticas avanzadas, y si algún equipo cambia significativamente (o al inicio de una nueva temporada), actualizar `equipos.txt` y `posiciones.txt` con los datos más recientes de equipo y defensa por posición.

En una implementación futura, se podría integrar APIs: - **API de datos NBA:** Consumir una API oficial o de terceros (por ejemplo, la NBA Stats API) para obtener las estadísticas de los partidos recientes automáticamente y poblar `partidos.csv` . Del mismo modo, obtener los promedios de jugadores (USG%, TS%) y rankings de equipo actualizados sin intervención manual. Esto reduciría el trabajo manual y aseguraría consistencia. Por ejemplo, la clase `NBADataLoader` podría extenderse para llamar a la API y actualizar los DataFrames en lugar de leer CSV. - **API de cuotas de apuestas:** Otra posibilidad es conectar a un servicio de cuotas deportivas en tiempo real para obtener las líneas y cuotas directamente. Así, en vez de pedir al usuario que ingrese las líneas manualmente, el sistema podría sugerir apuestas disponibles (ej. *"Over 22.5 puntos a cuota 1.90"*) para un jugador en un partido futuro y automáticamente saber la cuota. Esto haría el flujo más automatizado y facilitaría evaluar muchas apuestas rápidamente.

Es importante destacar que cualquier integración de API debe manejar los formatos de datos y la sincronización con el modelo. Por ejemplo, si obtenemos datos de nuevos partidos, habría que reentrenar o al menos actualizar el modelo con cierta periodicidad para incorporar esos partidos (en la implementación actual, cada vez que ejecutamos una predicción se entrena de cero con todos los datos; con muchos datos, tal vez convenga entrenar offline y solo cargar modelos).

Últimos 5-10 partidos: Muchos features clave (promedios móviles, descanso) miran la ventana de últimos juegos. Por eso, tener los últimos 5-10 partidos por jugador es crucial. Si el sistema no se mantiene al día, un jugador podría haber cambiado su rendimiento y el modelo no capturarlo. Por ejemplo, si un jugador estrella se lesiona o baja su minutos drásticamente, pero `partidos.csv` no contiene esos últimos partidos, el modelo seguiría prediciendo basado en datos antiguos. Mantener el *feed* de datos actualizado asegura que el modelo detecte tendencias actuales (subida o bajada de forma, cambios de roles, etc.).

En resumen, aunque el sistema es funcional con datos estáticos, su eficacia a largo plazo depende de la actualización continua. La incorporación de APIs de datos deportivos puede hacer el sistema más dinámico y cercano al tiempo real, permitiendo predicciones y recomendaciones siempre basadas en la información más reciente disponible. Esto es especialmente importante en el ámbito de apuestas deportivas, donde las condiciones cambian día a día (lesiones, trades, rachas de equipos, etc.).

5. Contenido y propósito de cada archivo de datos

El sistema se apoya en varios archivos de datos proporcionados, cada uno con un rol específico:

- **`partidos.csv`**: Es la base de datos histórica principal de rendimiento de jugadores por partido. Cada fila corresponde a la actuación de un jugador en un partido específico e incluye columnas como jugador, equipo rival, puntos, rebotes, asistencias, minutos jugados, estadísticas de tiro (FGM/FGA, 3PM/3PA, etc.), pérdidas, faltas, posición (`pos`), temporada (`temporada`), fecha del partido y si jugaba de local o visitante. Este archivo alimenta el entrenamiento del modelo: de aquí se sacan los valores objetivos (cuántos pts/reb/ast hizo realmente) y se calculan la mayoría de features basadas en historial (promedios móviles, último descanso, etc.). En esencia, `partidos.csv` es donde el modelo "aprende" las estadísticas pasadas para predecir las futuras. Es fundamental mantenerlo actualizado con nuevos partidos para reflejar la forma reciente de los jugadores.

- **`jugadores.txt`**: Contiene estadísticas avanzadas agregadas por jugador, probablemente promedios de la última temporada o carrera. Las columnas (definidas por `COLS_JUGADORES`) son:

- **jugador** (nombre),
- **USG_perc** (Porcentaje de Uso),
- **TS_perc** (True Shooting %),
- **AST_perc** (Assist Percentage),
- **REB_perc** (Rebound Percentage) [130].

Cada línea corresponde a un jugador de la NBA con esos valores. Por ejemplo, una línea puede ser: *"LeBron James,31.5,58.0,34.4,12.3"* indicando LeBron, 31.5% de uso, 58.0% TS, 34.4% AST%, 12.3% REB%. Estos datos proveen contexto de qué tipo de jugador es: si tiene alto USG y moderado TS, es un anotador que asume muchos tiros; un alto AST% indica función de base creador, etc. En el sistema, este archivo es leído y combinado para añadir esas columnas al feature set [59]. Si un jugador no está en `jugadores.txt` , el código lo detecta y asigna valores por defecto neutros [83]. Por tanto, es importante que `jugadores.txt` tenga al menos los jugadores relevantes. Se suele actualizar al inicio de temporada o cuando nuevos jugadores aparecen.

- **`equipos.txt`**: Lista los equipos NBA con métricas de equipo agregadas. Según `COLS_EQUIPOS` , las columnas incluyen:
- **TEAM** (código de equipo, p.ej. LAL, BOS),
- **GP** (posiblemente juegos o posesiones, el código ignora "POSS" que podría ser posesiones totales [131]),
- **OffRtg** (Offensive Rating, puntos por 100 posesiones anotados),
- **DefRtg** (Defensive Rating, puntos por 100 posesiones permitidos),
- **NetRtg** (OffRtg - DefRtg),
- **AST_perc** (% de asistencias del equipo),
- **AST_TO** (ratio asistencias/pérdidas),
- **REB_perc** (% de rebotes totales capturados),
- **eFG_perc** (efectividad de tiro ajustada),
- **TS_perc** (True Shooting % del equipo),
- **PACE** (posesiones por 48 minutos),
- **PIE** (Player Impact Estimate del equipo, una medida global de dominio),
- **POSS** (posesiones totales, que se elimina después de cargar) [132].

Cada fila es un equipo. Ejemplo: *"GSW,97.1,115.0,112.3,2.7,65.0,1.90,51.2,55.5,58.7,102.3,53.1"* (no real, ilustrativo). Estas estadísticas describen el estilo y fortaleza del equipo: un OffRtg alto y DefRtg bajo indica un equipo ofensivo y flojo en defensa, Pace alto indica juego rápido, etc. En nuestro sistema, al predecir para un jugador contra X equipo,

incorporamos las stats de X equipo desde aquí [27] . Así el modelo sabe “el rival tiene cierto perfil”. Mantener `equipos.txt` actualizado (al menos cada temporada, o a mitad de temporada) ayuda a reflejar si un equipo mejoró o empeoró.

- **`posiciones.txt`** : Contiene información defensiva por posición para cada equipo. Las columnas (según `COLS_POSICIONES`) son:
- **TEAM** (equipo),
- **pos** (posición, típicamente PG, SG, SF, PF, C),
- **ptspp** (puntos permitidos por partido a esa posición),
- **rebpp** (rebotes permitidos por partido a esa posición),
- **asistpp** (asistencias permitidas por partido a esa posición).

Cada equipo aparece repetido en 5 filas (una por posición). Por ejemplo:

```
LAL,PG,22.5,6.4,9.1
LAL,SG,18.3,5.0,4.3
...
LAL,C,15.7,11.8,2.1
```

Esto indicaría que Lakers permiten en promedio 22.5 puntos a bases rivales, 15.7 a pívots, etc. Este archivo es crucial para captar **matchups**: si un jugador es pívot y enfrenta a un equipo que permite muchos rebotes a los pívots, eso sube su expectativa de rebotes. El FeatureEngineer une por TEAM y pos del jugador para traer `ptspp` , `rebpp` , `asistpp` al feature set [133] . Si un equipo no tiene esa posición (por ejemplo, pos rara) o hay algún fallo, se maneja la excepción y se usan valores default [85] , pero idealmente `posiciones.txt` debería cubrir todos los equipos y posiciones estándar. Conviene actualizarlo cada temporada porque los patrones defensivos por posición pueden cambiar con nuevos jugadores o técnicos.

- **`apuestas_log_simple.csv`** : Es el **registro histórico de apuestas evaluadas** por el sistema. Sus columnas, como definidas al guardar apuesta [128] , son:
- `fecha` (fecha y hora en que se realizó la predicción),
- `jugador` , `rival` , `stat` (tipo de apuesta, ej. "pts"),
- `linea_apuesta` (la línea numérica, ej. 25.5),
- `cuota_ofrecida` (la cuota que daba la casa de apuestas para el over),
- `cuota_justa` (la cuota justa calculada por el modelo = 1/probabilidad_modelo),
- `probabilidad` (probabilidad estimada de éxito por el modelo, en [0,1]),
- `valor_esperado_EV` (el EV calculado = prob*cuota_ofrecida - 1),
- `recomendacion` (texto de recomendación dado, "APOSTAR" o "NO apostar" con sus criterios),
- `seguridad` (Alta, Media, etc. según la tabla de probabilidad/volatilidad),
- `acierto` (0/1, inicialmente 0 cuando se registra porque el partido aún no se ha jugado; el usuario debe actualizarlo manualmente a 1 si la apuesta fue ganadora una vez conocido el resultado),
- `proba_XGB` (la probabilidad estimada por el clasificador XGB de `cuotas.py` . Inicialmente queda vacía para nuevas entradas, y es rellenada cuando se ejecuta `cuotas.py` después).

Este archivo comienza vacío y va creciendo cada vez que usamos el sistema para evaluar apuestas. Sirve de base para el análisis de performance y para entrenar el modelo de `cuotas.py` . Por ejemplo, tras varias predicciones, podríamos ver filas como:

```
fecha           jugador         rival stat linea_apuesta cuota_ofrecida ...
acierto proba_XGB
```

```
2025-07-20 11:11 Lebron James    NYK  pts   25.5         1.85           ...
0       0.612 2025-07-20 11:11 Lebron James    NYK  reb    5.5         3.50
...
0       0.455
```

(Aquí se ilustran dos apuestas evaluadas el 20/07/2025 sobre Lebron). Podemos ver que el sistema registró probabilidad ~0.74 (74%) para cada una y EV positivos, recomendando quizás apostar. El usuario luego deja `acierto=0` hasta saber si se cumplieron (por ejemplo, si Lebron efectivamente pasó la línea de puntos o rebotes en el partido contra NYK; si sí, cambiaría acierto a 1). Después de ejecutar `cuotas.py` , aparecen `proba_XGB` = 0.612 y 0.455, que son las probabilidades que el clasificador de histórico asignó a esos overs respectivamente. Este log, por tanto, agrega una capa de aprendizaje: no solo guardamos lo que predijo el modelo, sino también el resultado, permitiendo refinar futuras predicciones.

En la práctica, **el usuario debería actualizar la columna "acierto"** tras cada partido para indicar si la predicción se cumplió (1) o no (0). El resto de columnas las rellena automáticamente el sistema. Es buena idea revisar este archivo periódicamente para analizar manualmente qué tipo de apuestas están funcionando mejor y también para detectar posibles sesgos (por ejemplo, si vemos sistemáticamente `acierto` =0 en cierto tipo de apuestas, convendría revisar el modelo para ese stat).

En conjunto, estos archivos aportan al sistema: `partidos.csv` la materia prima de entrenamiento, `jugadores/equipos/posiciones` enriquecen las características contextuales, y `apuestas_log_simple.csv` cierra el ciclo con el historial de predicciones para evaluación y reentrenamiento. Mantener cada uno correctamente formateado y actualizado es crucial para el rendimiento y la confiabilidad del predictor.

6. Guía práctica: Cómo utilizar el sistema paso a paso

Para un usuario técnico principiante que quiera usar este sistema de predicción de apuestas NBA, se detallan a continuación los pasos a seguir y consideraciones en cada etapa:

1. **Actualizar los datos de partidos y estadísticas**: Antes de realizar predicciones nuevas, asegúrese de que los archivos de datos estén al día:
2. Abra `partidos.csv` y agregue las filas de los partidos más recientes que no estén incluidos. Cada fila debe contener la actuación de un jugador en un partido, con la fecha formateada correctamente (YYYY-MM-DD) y los campos numéricos en su unidad (puntos, rebotes, asistencias como enteros, porcentajes como 0-100 en lugar de 0-1, etc. según el formato existente).
3. Si hay jugadores que debutaron recientemente o que cambiaron significativamente, puede ser útil actualizar sus datos en `jugadores.txt` (por ejemplo, si un jugador incrementó mucho su uso ofensivo en la nueva temporada, reflejarlo).
4. Actualice `equipos.txt` si ha iniciado una nueva temporada o si desea reflejar cambios (traspasos, lesiones prolongadas) que alteren sustancialmente el rendimiento de equipo.
5. Actualice `posiciones.txt` para la nueva temporada, ya que las tendencias de defensa por posición pueden cambiar. Esto suele hacerse con datos acumulados de la actual temporada.
6. *Nota:* Si realiza cambios importantes o estructura diferente en estos archivos, verifique que las columnas obligatorias sigan presentes (el sistema validará que no falten columnas requeridas [134]).
7. **Ingresar las apuestas a evaluar**: Ejecute el script principal del predictor. Si está en un entorno de desarrollo, esto suele ser correr `python predictor.py` en la terminal. El programa mostrará un menú con opciones:

8. Elija la opción **"1 - Realizar predicción"** [135] para empezar a evaluar una nueva apuesta. El sistema interactuará por consola solicitando:

   Jugador: Escriba el nombre del jugador tal como aparece en los datos (no distingue mayúsculas/minúsculas, pero sí asegúrese de la ortografía, por ejemplo "LeBron James"). *Rival*: Ingrese el código de 3 letras del equipo rival (franquicia) – por ejemplo, LAL para Lakers, BOS para Celtics. Debe coincidir con los códigos usados en `equipos.txt` y `partidos.csv` (verifique la nomenclatura, e.g. Los Angeles Lakers probablemente sea "LAL").

   Minutos estimados: Ingrese cuántos minutos de juego espera que dispute el jugador en ese partido. Este valor es importante ya que si un jugador normalmente juega 35 minutos pero estimamos que jugará solo 20 (por lesión o decisión táctica), las predicciones deberán ajustar a menor producción. Use un valor de tipo float (puede ser decimal).

   Local (s/n): Responda "s" si el partido es en casa del jugador (local), o "n" si juega de visitante. Esto se convertirá en 1/0 internamente.

   Último resultado del equipo (v/d): Indique si el equipo del jugador ganó (v) o perdió (d) su partido anterior. Recuerde que actualmente esto solo se guarda como dato pero el modelo no lo usa profundamente; aun así, es bueno registrarlo para potenciales mejoras futuras.

9. Tras ingresar estos datos, el sistema le pedirá ahora las **líneas de apuesta y cuotas**:

   Se solicita *"Over Puntos línea (ej. 25.5):"* – ingrese la línea de puntos que ofrece la casa de apuestas para el jugador. Por ejemplo, si la apuesta es over/under 25.5 puntos, escriba 25.5.

   Luego *"Cuota Over Puntos ofrecida (ej. 1.85):"* – ingrese la cuota (decimal) asociada al over de esa línea. Por ejemplo 1.85.

   El mismo proceso se repite para *Rebotes* y *Asistencias*. Si para alguna de esas categorías no le interesa hacer apuesta, igual debe ingresar algo (podría ingresar la línea y cuota referencial o 0 y 0, pero lo recomendable es si no va a evaluar, aún así poner la línea normal para que los cálculos se muestren; en caso de no tener cuota, podría poner 1.0 solo para completar, sabiendo que EV no tendrá sentido).

   Ejemplo: Supongamos queremos evaluar **Over puntos** de LeBron James vs NYK con línea 25.5 a cuota 1.85, y también nos intriga el **Over rebotes** con línea 5.5 a cuota 3.50.

   Ingresaríamos:

   Jugador: LeBron James

   Rival: NYK

   Minutos estimados: (por ejemplo) 34.0

   Local: s/n según corresponda (imaginemos juega de visitante, pondríamos "n")

   Último resultado: v/d según corresponda (supongamos viene de victoria, "v")

   Over Puntos línea: 25.5

   Cuota Over Puntos: 1.85

   Over Rebotes línea: 5.5

   Cuota Over Rebotes: 3.50

   Over Asistencias línea: (podemos ingresar 0 si no nos interesa, o alguna línea estándar solo para ver; pongamos 8.5)

   Cuota Over Asistencias: 1.85 (por ejemplo).

10. Con estos inputs, el programa procederá a ejecutar la predicción y mostrar resultados.

11. **Ejecutar la predicción y entender los resultados**: Una vez ingresados los datos, el sistema imprimirá en consola un reporte detallado:

12. Primero, una advertencia si el jugador tiene pocos partidos en la base (esto para que tomemos con precaución los números si n<10) [114] .

13. Luego las predicciones de **Puntos, Rebotes, Asistencias** que el modelo estima. Ejemplo: *"Puntos: 26.3 | Rebotes: 6.1 | Asistencias: 7.5"*. Estos son los valores esperados (medias) que calculó el modelo XGBoost, dados los minutos y contexto ingresados.
14. Se mostrará la **volatilidad** para cada stat. Siguiendo el ejemplo, podría salir: *"VOLATILIDAD (0.4+ = alta): Puntos: 0.28 | Rebotes: 0.35 | Asistencias: 0.25"*. Vemos que rebotes tiene 0.35 (>0.34 Moderada tirando a alta), puntos 0.28 (Media) y asistencias 0.25 (Baja). Esto nos indica en cuál categoría el jugador es más impredecible (rebotes en este caso).
15. A continuación, sección de **detalles de simulación (log-normal)**. Por cada tipo de apuesta:

    Se muestra la línea y la probabilidad de que el jugador la supere. Ej: *"PTS – Línea: 25.5 – Probabilidad Over: 74.0%"*. Esta probabilidad proviene de la simulación Monte Carlo [118]. En este ejemplo, el modelo estima que hay un 74% de chances de que LeBron anote 25.5 puntos.

    Cuota justa (modelo): Por ejemplo "1.35". Esto es simplemente 1/0.74 [118]. Compa[illegible]do: la cuota ofrecida era 1.85. Si la cuota justa modelo (1.35) es mucho menor que la ofrecida (1.85), significa que la apuesta tiene valor (paga mucho más de lo que debería estadísticamente).

    Cuota ofrecida: la que ingresamos, e.g. *"1.85"*.

    Valor Esperado (EV): calculado como `probabilidad * cuota_ofrecida - 1` [136]. En nuestro ejemplo, EV = 0.74*1.85 - 1 = 0.369, es decir +36.9%. Se mostrará como* "EV: 0.369"***** (podría redondear a 0.37). Esto indica un rendimiento esperado muy positivo (por cada 1€ apostado se espera ganar en promedio 0.37€). Si el EV fuera negativo, significaría que la apuesta no conviene estadísticamente.

    Seguridad: basada en la prob y volatilidad, en este caso prob alta y vol medio daría *"Alta"* seguridad [120]. Esto aparece[illegible]en la línea correspondiente.

    Luego, datos históricos: si hemos hecho apuestas similares antes, por ejemplo otras apuestas de PTS, se imprimirá cuántas hay, el porcentaje de acierto histórico en PTS, el EV medio histórico y el beneficio total acumulado en ellas [122]. Si en nuestro log no hay suficientes (o [illegible]guna) apuestas de ese tipo, dirá *"Sin datos históricos suficientes para evaluar recomendaciones"* [137].

    Recomendación: Basándose e[illegible]odo lo anterior (y las reglas mencionadas de EV>0 y buen histórico), el sistema mostrará *">> Recomendación: APOSTAR (EV positivo + buen histórico)"* en verde si corresponde [123]. Si, por ejemplo, el EV hubiera sido ne[illegible]ivo, veríamos *">> Recomendación: NO apostar (histórico no favorable o EV negativo)"* en rojo [124].

    Esta recomendación final le ayuda a decidir.

    Percentiles simulados: se listarán los valores de la estadística en percentil 5, 25, 50 (mediana), 75, 95 [138]. Ejemplo: *"Percer[illegible]s: 5%: 15 | 25%: 22 | 50%: 26 | 75%: 30 | 95%: 35"* puntos. En el 5% de los casos LeBron haría 15 o menos, en el 95% de los casos 35 o menos, etc. Esto da idea del rango de posibles resultados.

    Simulaciones realizadas: típicamente "10000" o quizá 15000 si la volatilidad era alta (el código aumenta simulaciones con alta volatilidad) [139] [140].
16. Lo anterior se repetirá para Rebotes y Asistencias. Siguiendo el ejemplo, para rebotes la salida podría ser: *"REB – Línea: 5.5 – Prob. Over: 74.0% – Cuota justa: 1.35 – Cuota ofrecida: 3.50 – EV: 1.595 – Seguridad: Media – Histórico: ... – >> Recomendación: APOSTAR"*. Notamos aquí un caso interesante: cuota 3.50 con 74% prob es un EV extremadamente alto (~+160%). Esto normalmente no ocurriría en cuotas reales (sería una arbitraje claro), pero supongamos que fue un ejemplo llamativo. El sistema obviamente recomendaría apostar ya que hay mucho valor.

17. Tras mostrar los resultados para PTS, REB, AST, se completa el output.

18. **Registro y uso del historial**: Cada vez que obtiene un resultado, el sistema **guarda automáticamente** esa apuesta en `apuestas_log_simple.csv` mediante la función `guardar_apuesta` [128] [129]. No necesita editar nada para registrarla (excepto posteriormente anotar el acierto). Después de ejecutar una predicción, se recomienda:

19. **Anotar el resultado real**: Una vez finalizado el partido, abra `apuestas_log_simple.csv` (puede ser en Excel, Google Sheets o un editor de texto) y edite la columna *acierto* de las apuestas correspondientes: ponga "1" si la apuesta resultó ganadora (es decir, el jugador superó la línea) o "0" si falló. Esto es importante: el sistema no sabe el resultado real automáticamente, requiere que usted actualice esa columna manualmente.

20. Guardar el archivo log con los cambios. Ahora su histórico está actualizado.

21. **Re-entrenar el modelo de cuotas (opcional/periódico)**: Si ya acumuló al menos 100 apuestas en el log (o en general tras añadir nuevos resultados), puede ejecutar `cuotas.py` para recalcular la probabilidad ajustada *proba_XGB* en su log.

22. Correr `python cuotas.py` en la terminal. Este script leerá todo `apuestas_log_simple.csv` , construirá los features históricos y entrenará el XGBClassifier si hay suficientes datos [35]. Al finalizar, imprimirá las probabilidades XGB de las apuestas del día (si usted recién agregó los resultados de ayer, esas serán las últimas entradas).
23. Revise en la salida las `proba_XGB` . Si por ejemplo nuestra apuesta de LeBron puntos apareciera con proba_XGB = 0.60, significa que según el modelo de histórico había 60% de chances de que se acertara (en comparación al 74% que creíamos). Esto podría sugerir que quizás el modelo principal fue demasiado optimista, o que históricamente apuestas similares se cumplen el 60% del tiempo.
24. El archivo `apuestas_log_simple.csv` ahora también tendrá la columna *proba_XGB* rellenada para todas las apuestas desde la #100 en adelante. Puede abrirlo para ver esta columna. No se sorprenda si las primeras 100 filas tienen proba_XGB = 0, ya que el sistema no las usó para entrenar inicialmente.

25. *Frecuencia:* No es necesario correr `cuotas.py` cada vez que haga una nueva predicción. Pero hacerlo de vez en cuando (por ejemplo, cada 10-20 apuestas nuevas) permitirá que el modelo de calibración se mantenga actualizado. Sobre todo, cada vez que haya actualizado varios "acierto" después de una tanda de partidos, es buen momento para re-entrenar.

26. **Refinamiento continuo:** A medida que utilice el sistema en el tiempo:

27. Continúe ampliando el dataset de partidos. Quizá anualmente, re-entrene el modelo XGBoost principal si siente que las tendencias han cambiado mucho (en la implementación actual, el modelo se entrena desde cero cada ejecución con todos los datos disponibles, así que realmente siempre está "reentrenado" hasta la última información).
28. Use el histórico de apuestas para aprender también manualmente: la función de análisis (opción 2 del menú principal) le mostrará su porcentaje de aciertos global y EV medio [141]. Un porcentaje de ac o global inferior al requerido por las cuotas medianas (por ejemplo, si acierta 50% pero suele apostar cuotas 1.85, que requieren ~54% para break-even) indicaría que hay que ajustar la estrategia o mejorar el modelo.
29. Fíjese en el desglose por tipo de apuesta que imprime el análisis (en `predictor.py` al final de la opción 2, calcula acierto por stat y EV medio por stat) [142] [143]. Tal vez descubra que, hipotéticamente, en rebotes tiene 70% acierto y EV positivo, pero en asistencias solo 40% acierto y EV negativo. Eso sugeriría que las predicciones de asistencias no están bien calibradas (podría ser necesario revisar las features o ser más conservador con asistencias).
30. Con el clasificador de cuotas (`proba_XGB`), a futuro se podría incluso incorporar su salida como input para la recomendación. Por ejemplo, solo apostar si tanto el modelo principal como el modelo histórico están de acuerdo en que hay valor. Por ahora, usted puede hacer esto manualmente: si el modelo principal dice "Apostar" pero *proba_XGB* sale baja (digamos <0.5), tal vez conviene dudar y analizar por qué hay discrepancia.
31. **Mantenimiento de archivos:** Guarde respaldos de sus CSV periódicamente. Especialmente `apuestas_log_simple.csv` es valioso porque contiene la evidencia de rendimiento. También, antes de

iniciar una nueva temporada NBA, puede reiniciar el log o marcar de alguna forma las temporadas (añadiendo columna temporada al log quizá), ya que el comportamiento puede cambiar año a año.

Siguiendo estos pasos, podrá utilizar el sistema de forma efectiva. En resumen, el flujo típico por apuesta es: actualizar datos -> ingresar apuesta en el predictor -> obtener predicción y recomendaciones -> apostar (si decide hacerlo) -> tras el partido, actualizar resultado en log -> (opcionalmente) correr análisis/cuotas.py -> repetir. Con el tiempo, el sistema debería ayudarle a identificar buenas apuestas y evitar las de valor negativo, mejorando sus decisiones basadas en datos.

7. Ejemplo práctico con datos reales

Para ilustrar todo lo anterior, tomemos un ejemplo práctico usando datos reales cargados en el sistema:

Ejemplo: Supongamos que queremos evaluar la apuesta **"Over 25.5 Puntos" de LeBron James contra los New York Knicks**, con una cuota ofrecida de 1.85. Además, estamos considerando **"Over 5.5 Rebotes" del mismo jugador en ese partido**, con cuota 3.50. Vamos a recorrer cómo el sistema procesaría esta solicitud paso por paso, apoyándonos en los datos existentes:

- **Situación previa (datos recientes):** Del archivo de partidos, LeBron James tiene histórico amplio (140+ partidos en las temporadas 23/24 y 24/25). Sus últimos 10 partidos promedia alrededor de 21.9 puntos y 5.8 rebotes21†. Sin embargo, contra el rival específico (NYK), en sus últimos 4 enfrentamientos disponibles promedió ~28.3 puntos22†, indicando que tiende a rendir bien ante New York. Los Knicks, según `equipos.txt` , tienen un DefRtg de 113.3 (medio) y Pace ~97.6 (bastante bajo, tienden a juegos más lentos). En `posiciones.txt` , para la posición de LeBron (alero, SF) contra NYK, figura que permiten ~23.0 puntos, 7.6 rebotes y 4.4 asistencias en promedio a los aleros rivales. LeBron tiene un USG% de 29.6 y TS% ~58% según `jugadores.txt` (lo que refleja su gran protagonismo ofensivo con buena eficiencia). Todos estos datos se entrelazan en las features.

- **Predicción del modelo XGBoost:** Considerando los minutos (digamos estimamos jugará ~34 min) y que viene de una victoria, el modelo integrará:
- Su promedio reciente (21.9 pts) pero ajustará al alza por el buen desempeño vs Knicks (28.3 pts) y porque Knicks permiten bastante a su posición (~23 pts, por encima de lo que muchos equipos permiten).
- El ritmo bajo de Knicks podría moderar un poco la predicción (menos posesiones => ligeramente menos oportunidades).
- Sus altos USG% sugieren que si el juego es competitivo, él tomará muchos tiros.
- Tras procesar todo, el modelo podría dar una predicción cercana a **26 puntos** para LeBron en ese juego (hipotético resultado de la XGBoost). Para rebotes, quizá prediga alrededor de **6-7 rebotes**.

En efecto, supongamos que el sistema imprime: *"Puntos: 26.1 | Rebotes: 6.4 | Asistencias: 7.0"* como predicción.

- **Cálculo de volatilidad:** LeBron tiene una larga trayectoria, suele ser bastante consistente en puntos, un poco más variable en rebotes. Supongamos el cálculo resultó en volatilidades: *pts: 0.26, reb: 0.32, ast: 0.20*. Esto indica un coeficiente de variación ~26% en puntos (bastante bajo para un anotador élite, confiable), ~32% en rebotes (un poco más volátil, dependiente del juego), y asistencias muy consistente (20%). Se marcaría "Alta" volatilidad si >0.40, no es el caso en ninguno. Son "Moderada" en reb, "Baja" en ast, "Moderada-Baja" en pts.

- **Simulación Monte Carlo:** Usando la predicción de 26.1 pts y vol 0.26, la distribución log-normal se genera. La probabilidad de superar **25.5** puntos se calcula. Dado que 25.5 está apenas por debajo de la media (26.1), y con volatilidad moderada, la simulación podría dar alrededor de un **70-75%** de casos por encima de 25.5. Imaginemos que fue **74%** [118], lo cual concuerda con la media ligeramente por encima de la línea. Para rebotes, media 6.4 vs línea 5.5, probabilidad también alta (por ejemplo 74% según el ejemplo dado, quizás reflejando

que 5.5 es baja comparada con su media, aunque la cuota indica que la casa lo ve improbable, en nuestro caso el modelo discrepa).

- **Resultados presentados:** El sistema mostraría algo como:

- **PTS:** Línea 25.5, Probabilidad Over ~74%, Cuota justa ~1.35 [119]. Cuota ofrecida 1.85, EV 0.74*1.85 - 1 = +0.369 (+36.9%) [136]. Seguridad "Alta" (prob >% y vol<0.28 cumple los criterios)
  .

  Histórico: supongamos en el log tenemos 10 apuestas de tipo puntos, de las cuales 6 aciertos (60%) con EV medio +0.10. Esto se imprimiría: *"Histórico apuestas pts: 10 apuestas – % acierto histórico: 60.0% – EV medio histórico: 0.100 – Beneficio acumulado: X"*. Este historial es decente (por encima del 54% requerido para salir tablas en cuotas ~1.85).

  Recomendación: Con EV positivo + historial >50%, el sistema dirá **APOSTAR** [123]. Percenti podría mostrar que en 5% de simulaciones anotó 18 pts, mediana ~26 pts, 95% 34 pts, etc., dando confianza de que 26 es un valor central.
- **REB:** Línea 5.5, Prob Over ~74% también en este ejemplo. Cuota justa 1.35 vs ofrecida 3.50 significa EV enorme +1.59 [136] (159% ROI esperado sto es inusual; probablemente en la realidad una cuota 3.50 implicaría solo ~28% prob, pero nuestro modelo sugiere está muy subestimada.

  Seguridad "Media" (la prob es >57% pero la volatilidad de rebotes 0.32 está por encima de 0.28, y además cuota tan alta sugiere evento menos seguro) [120].

  Histórico: quizá solo pocas apuestas de rebotes en el log, digamos 3, con 2 aciertos (66%). No es suficiente para estar seguros, el sistema quizás diga "Sin datos históricos suficientes..." o muestre los que haya.

  Recomendación: Aun sin mucho histórico, EV es tan positivo que seguramente indicará **APOSTAR (EV positivo + buen histórico)** en verde.

  Percentiles de rebotes: 5%: 2 reb, 25%: 5 reb, 50%: 6 reb, 75%: 8 reb, 95%: 11 reb. Esto sugiere que aunque la mediana es 6 (sobre la línea), hay un 25% de casos donde quedaría 5 (bajo la línea). De ahí la cuota alta, pero nuestro modelo cree más en la parte alta de la distribución.

Estos resultados concuerdan con el output de ejemplo en el log, donde para Lebron vs NYK se registraron 74% prob tanto en pts como reb. El sistema guarda estas predicciones en el log con *acierto=0* inicialmente [144].

Después del partido: Imaginemos que LeBron efectivamente anotó 30 puntos y tomó 6 rebotes. La apuesta de puntos ganó (acierto=1) y la de rebotes también (6 > 5.5, acierto=1). El usuario actualizaría esos valores en `apuestas_log_simple.csv`. Luego, al correr `cuotas.py` tras unos cuantos partidos más, el clasificador notaría que apuestas similares de rebotes altas dieron sorpresa (podría ajustar futuras probabilidades) y que en general nuestras predicciones vienen bien. Por ejemplo, podría asignar *proba_XGB* = 0.65 para overs de puntos como el que hicimos (dándonos confianza adicional) y quizá 0.50 para overs de rebotes tan arriesgados (se cumplió esta vez, pero no siempre confía al 74%). Con más datos, afinará estos juicios.

En conclusión, el ejemplo demuestra cómo el sistema combina **datos estadísticos** (promedios, tendencias) con **algoritmos avanzados** (XGBoost, Monte Carlo) para ofrecer una estimación informada de una apuesta. Nos indicó que el over de 25.5 puntos de LeBron tenía alta probabilidad (y en efecto sucedió), y detectó un posible infravalor en rebotes (que también se cumplió). Si estos éxitos se repiten consistentemente, el sistema ayudará a identificar apuestas rentables. Lo importante para el usuario es seguir el proceso: alimentar con datos actualizados, interpretar las salidas (no cegarse solo con la probabilidad sino ver volatilidad y contexto), y usar las herramientas de histórico para ajustar sus estrategias de apuesta en el futuro. En manos de un analista, este manual y sistema permiten una **apuesta deportiva basada en datos y rigor técnico**, más allá de la intuición.