ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

Curso Académico 2024/2025

Trabajo Fin de Grado

OPTIMIZACIÓN Y EVALUACIÓN DE MODELOS DE CLASIFICACIÓN PARA DETECCIÓN DE TRAMPAS EN VIDEOJUEGOS MULTIJUGADOR ONLINE

Autor: Pablo Prior Molina

Directores: Jesús Sánchez-Oro Calvo
Sergio Pérez Peló

Índice

Índice de Figuras

1. Resumen

Este Trabajo de Fin de Grado tiene como objetivo principal el diseño, optimización y evaluación de modelos de clasificación para la detección automática de trampas (*cheating*) en videojuegos multijugador online. Para establecer un contexto, las prácticas desleales en videojuegos perjudican notablemente la experiencia de juego, por lo que la identificación temprana de usuarios tramposos mediante el uso de técnicas de **Machine Learning** representa una alternativa efectiva y escalable en contrapartida a otros sistemas tradicionales basados en reglas estáticas o revisiones manuales.

Para este objetivo, se han aplicado y comparado tres algoritmos de clasificación sumamente utilizados en entornos con datos tabulares: **Random Forest, Gradient Boosting y XGBoost**. Estos modelos han sido entrenados y evaluados en base a un conjunto de datos públicos derivados del trabajo *"Identify As A Human Does: A Pathfinder of Next-Generation Anti-Cheat Framework for First-Person Shooter Games"*. El *dataset* original fue sometido a un procedimiento de preprocesado que abarcó desde estandarización de variables, pasando por selección de características, hasta el tratamiento del desbalanceo mediante la técnica **SMOTE** (*Synthetic Minority Over-sampling Technique*).

La búsqueda de hiperparámetros se realizó utilizando tres estrategias complementarias: **Grid Search, Randomized Search y búsqueda manual**. Se procedió a evaluar los modelos utilizando métricas como **Accuracy, Precision, Recall y F1-Score**, con especial detenimiento y atención al comportamiento frente a clases desbalanceadas. Los resultados demuestran que el uso de técnicas de balanceo como SMOTE mejora significativamente la capacidad de detección de la clase minoritaria (jugadores tramposos), especialmente en modelos como **Gradient Boosting**, que llegó a alcanzar un **F1-Score** superior al 0.94.

Finalmente, se validaron los modelos más prometedores sobre un conjunto de test independiente del usado para el entrenamiento, indagando además en los efectos del ajuste del umbral de clasificación sobre el rendimiento general. Este trabajo pone de manifiesto la posibilidad y viabilidad de aplicar técnicas avanzadas de aprendizaje automático a la detección de comportamientos anómalos en entornos complejos como los videojuegos multijugador online usando de lenguaje **Python** debido a su reconocimiento y facilidad dentro de la industria.

2. Introducción

Este segundo capítulo persigue, entre otros objetivos, contextualizar el Trabajo de Fin de Grado, dar cuenta de la importancia tanto desde el prisma académico como profesional y establecer la delimitación del trabajo y de los conceptos relacionados. A su vez, se incluirá una revisión del estado del arte que sirva de punto de partida teórico, así como una descripción de cómo se va a articular el trabajo a nivel general.

En primer lugar, se dará un contexto donde se explique el problema de las trampas (*cheating*) en los videojuegos multijugador, describiendo por qué son un verdadero reto técnico y ético en lo que respecta a la industria del desarrollo de videojuegos. En segundo lugar, se indicará el enfoque que se ha llevado a cabo y los límites del trabajo. Tercero, se dedicará un espacio a los trabajos anteriores y a las técnicas similares, contrastando los trabajos que se han podido desarrollar. Finalmente, se mostrará una sencilla guía sobre la estructura de los capítulos posteriores.

2.1. Contexto y alcance

En los videojuegos multijugador actuales, más concretamente en títulos competitivos online, la solidez y el comportamiento del entorno de juego es fundamental para garantizar una experiencia equilibrada y justa. Aun así, la aparición de trampas (*cheats*) en las propias partidas por parte de algún jugador supone un desafío continuo para las empresas, desarrolladores y la comunidad de jugadores en general.

El ecosistema de los videojuegos ha cambiado en los últimos años perfilando un modelo centrado en la conectividad, el multijugador y la competitividad online. Comunidades gigantes como *Call of Duty, Counter-Strike, Valorant o Fortnite* demandan un *gameplay* que reúna las condiciones de justicia, seguridad y competitividad. Sin embargo, esta competitividad ha propiciado que existan determinados comportamientos fraudulentos, conocidos como *cheats* o trampas, que tienen como finalidad alterar el equilibrio de las partidas en favor de aquel jugador que los utiliza.

Estas trampas pueden presentarse de diferentes maneras: desde *aimbots* que automatizan la mejora de la puntería, *wallhacks* que permiten ver a través de las paredes, hasta incluso la modificación de la velocidad del movimiento, la manipulación de los paquetes de la red o la modificación del código del cliente. En muchos casos, los sistemas tradicionales de detención como los basados en firmas o reglas predefinidas no son suficientes para poder hacer frente a las trampas que intentan pasar por comportamientos humanos para no ser detectadas.

El problema se amplifica cuando se entra en el ámbito de los *e-Sports* (juegos competitivos) y en competiciones más serias donde los premios económicos y el reconocimiento profesional elevan la presión por sacar el mayor partido posible del juego, pero incluso en los ambientes casuales el uso de trampas hace que la experiencia del juego se degrade, erosionando así la base de usuarios, además de minar también la confianza hacia los desarrolladores y las plataformas.

La detección y prevención de trampas ha sufrido un cambio en cuanto a su actuación y aplicación. Originariamente surge como algo más manual y la disposición de análisis posteriores a la aparición de estos comportamientos sospechosos. Sin embargo, con el surgimiento de la inteligencia artificial y el análisis de datos que arrojan, aparece la posibilidad de emplear algoritmos de aprendizaje automático para identificar este tipo de patrones extraños en tiempo real, lo que supone poder tener más variantes y posibilidades de herramientas de detección para afrontar este tipo de problemas.

Este trabajo se centra en la aplicación de modelos de clasificación supervisada para abordar este problema. El objetivo principal es evaluar distintas arquitecturas de clasificación para determinar cuáles ofrecen el mejor equilibrio entre precisión y sensibilidad en la detección de trampas. Para ello, cabe resaltar una vez más, que se ha utilizado un *dataset* etiquetado con datos derivados del paper *"Identify As A Human Does: A Pathfinder of Next-Generation Anti-Cheat Framework for First-Person Shooter Games"* (Zhang et al., 2023), que proporciona información sobre comportamientos sospechosos y legítimos.

A lo largo del proyecto, se ha seguido un enfoque basado en la experimentación controlada. Se han probado distintos modelos como *Random Forest*, *Gradient Boosting* y *XGBoost*, utilizando tanto datos originales como datos balanceados con la técnica **SMOTE** (*Synthetic Minority Over-sampling Technique*). Se ha prestado especial atención al problema del desbalanceo de clases, común en estos contextos, donde los casos positivos (jugadores tramposos) son significativamente menos frecuentes que los negativos. Esta situación puede llevar a modelos que aparentan tener alto rendimiento (alta exactitud), pero que en realidad ignoran completamente la clase minoritaria.

Se ha optado principalmente por centrar el esfuerzo en la parte de modelado, optimización y validación de soluciones basadas en datos reales. Esto permite extraer conclusiones sólidas sobre el potencial del aprendizaje automático en este campo, permitiendo asentar una base para futuras investigaciones más complejas o integradas en entornos reales.

En resumen, este proyecto busca:

- Entender la problemática técnica y ética asociada al *cheating* en videojuegos multijugador.
- Explorar distintas técnicas de clasificación supervisada para predecir comportamientos tramposos.
- Evaluar el impacto del balanceo de clases en el rendimiento de los modelos.
- Ofrecer recomendaciones sobre el uso de técnicas de IA en sistemas *anticheat* modernos.

2.2. Estado del arte

Como se ha comentado, en los últimos años, la expansión exponencial de los videojuegos multijugador online ha sido acompañada con un incremento notable de las prácticas deshonestas por parte de algunos jugadores, lo que ha motivado a la industria a implementar técnicas cada vez más complejas para detectar trampas. Este escenario ha hecho que la aplicación de técnicas de aprendizaje automático se presente como una alternativa muy interesante para detectar comportamientos anómalos de manera activa. No obstante, estos sistemas se enfrentan igualmente a problemas como el desbalance de los datos y la necesidad de clasificaciones sólidas. A continuación, se revisan los principales enfoques para la tarea de detección de trampas aplicando los clasificadores más apropiados en esta área y técnicas para mitigar el impacto del desbalance.

2.2.1 Detección de trampas en videojuegos

La detección de trampas en videojuegos multijugador ha cobrado una importancia crítica en la industria del desarrollo de videojuegos. Existen numerosos enfoques que permiten prevenir y abolir este tipo de problemas. Recientemente, se ha incrementado el uso de métodos mediante aprendizaje automático para detectar este tipo de patrones sospechosos (Willman, 2020).

Estos sistemas generalmente se entrenan con conjuntos de datos etiquetados que contienen tanto comportamientos normales como tramposos, extrayendo características relevantes como el número de muertes, precisión de disparo, tasa de movimiento, velocidad de apuntado, entre muchos otros (Kotkov, 2021).

2.2.2 Clasificadores

Existen numerosos y populares métodos de clasificación que sirven para este tipo de tareas, por ello mismo, se han elegido tres de estos para profundizar y utilizar en este trabajo:

- **Random Forest:** Un ensamble de árboles de decisión que mejora la generalización del modelo mediante el uso de múltiples árboles entrenados con subconjuntos aleatorios de los datos y características. Es robusto frente a sobreajuste y adecuado para conjuntos de datos con alta dimensionalidad (Breiman, 2001).
- **Gradient Boosting:** Técnica que crea modelos secuenciales donde cada nuevo modelo corrige los errores cometidos por los anteriores. Suele proporcionar alta precisión en clasificación, aunque puede ser más sensible a los hiperparámetros (Friedman, 2001).
- **XGBoost:** Una mejora de *Gradient Boosting* que incorpora regularización L1/L2, manejo eficiente de valores faltantes y paralelismo. Ha sido ampliamente utilizado en

competiciones de ciencia de datos debido a su alto rendimiento y flexibilidad (Chen & Guestrin, 2016).

2.2.3 Desbalanceo de clases

En problemas como la detección de trampas en videojuegos, lógicamente, el número de jugadores y usuarios tramposos suele ser notablemente menor al de jugadores limpios y libres de estos inconvenientes. Esto puede provocar que los modelos que se utilizan queden sesgados hacia la clase mayoritaria (no tramposos), haciendo así que se obtenga una alta precisión, pero bajo *recall* para la clase minoritaria (tramposos) (Fernández et al., 2018).

2.2.4. SMOTE

SMOTE (*Synthetic Minority Over-sampling Technique*) es una técnica para abordar el desbalance creando ejemplos sintéticos de la clase minoritaria. En lugar de simplemente replicar datos, genera nuevas muestras interpolando entre vecinos más cercanos de la clase minoritaria (Chawla et al., 2002). Esto permite al modelo aprender mejor la frontera de decisión y mejora métricas como el *recall* y *F1-Score*.

2.3. Estructura del documento

Este Trabajo de Fin de Grado se ha organizado en siete capítulos principales, de los que ya se han visto dos de ellos (Resumen e Introducción). Todos estos capítulos abordan una parte esencial y participativa del desarrollo del proyecto, desde la motivación inicial hasta las conclusiones finales. A continuación, se describe brevemente el contenido de cada de los restantes:

- **Capítulo 3 – Descripción informática**: Este capítulo detalla el entorno de desarrollo utilizado (entorno virtual, librerías, versiones, IDE, etc.) y el flujo completo de implementación: carga y preprocesado de datos, selección de características, entrenamiento de modelos y estrategias de optimización de hiperparámetros. Se incluyen también detalles sobre el conjunto de datos y cómo se ha manipulado.
- **Capítulo 4 – Descripción algorítmica**: En este apartado se introducen los conceptos teóricos que sustentan el trabajo. Se explican los fundamentos del aprendizaje supervisado, la problemática del desbalanceo de clases y las técnicas utilizadas para su mitigación, como **SMOTE**. También se justifica la elección de los modelos seleccionados y se describe a alto nivel el enfoque comparativo adoptado.

- **Capítulo 5 – Resultados**: Se exponen y analizan los resultados obtenidos en las diferentes fases del experimento. Se comparan los modelos bajo distintos escenarios (con y sin SMOTE), se evalúan sus métricas de rendimiento y se discute el fenómeno del sobreajuste. Este capítulo representa el núcleo del trabajo empírico desarrollado.
- **Capítulo 6 – Conclusiones y trabajo futuro**: Resume los hallazgos más importantes del trabajo, reflexiona sobre las decisiones tomadas y plantea posibles líneas de mejora o continuidad para futuras investigaciones relacionadas.
- **Capítulo 7 – Anexos y bibliografía**: Contiene tanto las referencias bibliográficas utilizadas a lo largo del trabajo, presentadas en formato APA, como material adicional relevante.

3. Objetivos

3.1. Objetivo principal

En este capítulo se exponen los objetivos abordados en este Trabajo de Fin de Grado, desglosados en el objetivo principal y los objetivos secundarios.

3.2. Objetivos secundarios

- Preprocesar y equilibrar el conjunto de datos mediante técnicas como la estandarización y el sobremuestreo (**SMOTE**).
- Aplicar métodos de selección de características para reducir la dimensionalidad y mejorar el rendimiento de los modelos.
- Diseñar y comparar múltiples modelos de clasificación: **Random Forest, Gradient Boosting y XGBoost**.
- Realizar ajustes de hiperparámetros tanto mediante búsqueda manual como algoritmos **Random Search** y **Grid Search**.
- Evaluar el rendimiento de los modelos con métricas como precisión, **accuracy**, **recall**, **F1-score**.
- Comparar el rendimiento de los modelos entrenados con y sin **SMOTE** para valorar el impacto del equilibrio de clases.
- Seleccionar el modelo óptimo y aplicarlo al conjunto de test para analizar su desempeño final.

4. Descripción algorítmica

La metodología seguida en este trabajo se basa y refleja en un pipeline típico de un proyecto de aprendizaje automático, obviamente adaptado y adecuado a un contexto de trampas en videojuegos multijugador online. A continuación, las fases:

4.1. Obtención y preparación de datos

- **Dataset:** Se utilizó un conjunto de datos (*dataset*) extraído de un artículo llamado *"Identify As A Human Does: A Pathfinder of Next-Generation Anti-Cheat Framework for First-Person Shooter Games"* del Departamento de Ciencias de la Computación de la Universidad de Hong Kong. En este artículo se proporciona un enlace de descarga para un archivo que contiene estos conjuntos de datos, información del artículo, código entre otros archivos. En él se centra en la profundización de este tema sobre un famoso videojuego multijugador online llamado *Counter-Strike*.
- **Preprocesado:**

  Antes del entrenamiento de los modelos, se llevaron a cabo los siguientes pasos:

  - Eliminación de columnas irrelevantes (como el ID de usuario).
  - Escalado de características mediante *StandardScaler*.
  - Selección de características con *SelectKBest*, manteniendo las 7 más relevantes.
  - División del conjunto de entrenamiento en subconjuntos de entrenamiento y validación (80%-20%).

4.2. Balanceo de clases

- Para solucionar el desequilibrio de clases, que haya pocos tramposos, se aplicó la técnica de sobre muestreo **SMOTE** (*Synthetic Minority Over-sampling Technique*) en una parte del proyecto.
- Además, se mantuvieron modelos entenados sin aplicar **SMOTE** para comparar la influencia e impacto del balanceo en los resultados

4.3. Descripción del conjunto de datos

El conjunto de datos empleado en este proyecto proviene del trabajo de Zhang et al. (2023), titulado *"Identify As A Human Does: A Pathfinder of Next-Generation Anti-Cheat Framework for First-Person Shooter Games"* (ver referencia completa en la bibliografía). Este estudio propone un sistema avanzado de detección de trampas en videojuegos tipo **FPS** (*First-Person Shooter*), y como parte de su trabajo, los autores han publicado un conjunto de datos anonimizado que fue empleado en este Trabajo de Fin de Grado para el desarrollo, entrenamiento y evaluación de los modelos de clasificación.

Estructura del conjunto de datos

El *dataset* original incluye una serie de características estadísticas obtenidas del comportamiento de jugadores en un entorno del videojuego *Counter-Strike*. Cada muestra representa una sesión de juego, y contiene métricas que reflejan acciones como el tiempo de reacción, número de disparos, precisión, número de muertes, asistencias, entre otros.

El conjunto consta de tres archivos principales:

- *final_train.csv*: datos de entrenamiento y validación.
- *final_test.csv*: datos de evaluación final (test set).
- *final_validation.csv*: (no utilizado en esta implementación).

El conjunto de datos incluye además una variable objetivo, denominada *isCheater*, que indica si el jugador es considerado un tramposo (1) o no (0). Por tanto, consta de una clasificación binaria.

Distribución de clases

El *dataset* original presentaba un claro desbalance en la clase objetivo *isCheater*, con una desviación significativamente mayor de jugadores no tramposos frente a tramposos. Este desequilibrio fue una parte crítica en el diseño y entrenamiento de los modelos mencionados, por el hecho de que podía producir que los clasificadores tendieran hacia la clase mayoritaria de no tramposos.

Para mitigar este problema, se aplicó la técnica de sobremuestreo **SMOTE** (*Synthetic Minority Over-sampling Technique*), que permitió generar ejemplos sintéticos de la clase minoritaria para equilibrar el conjunto de entrenamiento.

Estas etapas fueron fundamentales para asegurar que los modelos trabajasen con datos limpios, normalizados y reducidos a las variables más significativas.

4.4. Preprocesado y transformación de datos

Antes de entrenar los modelos de clasificación, era importante aplicar una serie de cambios al conjunto de datos con el objetivo de mejorar la calidad del entrenamiento y asegurar una buena y correcta generalización.

4.4.1. Estandarización

Se aplicó una normalización usando *StandardScaler* de la librería *Scikit-learn*, la cual modifica cada variable para que tenga media 0 y desviación estándar 1. Esta transformación es muy relevante en modelos basados en árboles como **Gradient Boosting** y **XGBoost**, ya que, aunque en principio no se necesite, se observó una pequeña mejora en la estabilidad de los resultados, concretamente tras aplicar **SMOTE**.

4.4.2. Selección de características

Para reducir la dimensionalidad del *dataset* y centrarse en las variables más informativas o significativas, se aplicó una selección de características con *SelectKBest* utilizando el test *ANOVA F* (función *f_classif*). Se seleccionaron las 7 variables con mayor puntuación estadística frente a la variable objetivo. Esto permitió disminuir el ruido y acelerar tanto el entrenamiento como la validación de los modelos que se poseían.

4.4.3. Balanceo del dataset

Uno de los principales inconvenientes identificados en el análisis y desarrollo fue el desbalanceo de clases, donde los jugadores etiquetados como tramposos representaban una cifra bastante menor que los jugadores limpios.

Para tratar esto, se exploraron dos enfoques bien diferenciados:

- **Entrenamiento sin balanceo:** se usaron directamente los datos originales, aplicando únicamente técnicas como *class_weight='balanced'* en los modelos compatibles (**Random Forest**).
- **Entrenamiento con SMOTE:** se aplicó la técnica **SMOTE (Synthetic Minority Over-sampling Technique)** gracias a la librería *imbalanced-learn*, que añade nuevas muestras sintéticas de la clase minoritaria basadas en los vecinos más cercanos. Esto permitió entrenar modelos con un conjunto más equilibrado y evitar esa gran diferencia entre clases, y se observó un aumento considerable del *recall*, aunque en algunos casos afectando a la precisión.

4.4.4. División en conjuntos de datos

La partición de los datos se realizó de la siguiente manera:

- En el caso **sin SMOTE**, el *dataset* original se dividió en un **70% para entrenamiento, un 15% para validación y un 15% para test final**, asegurando así que los datos del test permanecieran completamente desconocidos durante todo el proceso de optimización para tener un correcto desarrollo.

- Para los experimentos **con SMOTE**, se aplicó primero la técnica de sobremuestreo sobre los datos de entrenamiento y validación (es decir, el **85% original**), y luego se realizó una nueva división interna: **80% para entrenamiento y 20% para validación**. Esta estrategia garantizó que el test final no se viera afectado por el balanceo artificial y que el entrenamiento se realizara sobre datos equilibrados pero realistas y así garantizar cierta coherencia.

4.5. Justificación de usos

Respecto al tema que se está tratando como es la detección de trampas en videojuegos multijugador online, se puede encasillar y concretar el problema como un trabajo o tarea de clasificación supervisada, en la cual se trata de averiguar y determinar si un jugador es tramposo o por el contrario no es tramposo a través de un conjunto de métricas extraídas durante las partidas de estos mismos. Este tipo de tarea se contempla a la perfección en el marco del aprendizaje automático supervisado, ya que se parte de datos etiquetados y se pretende dar con un modelo predictivo que pueda generalizar un conjunto de datos nuevo o en su lugar, un conjunto no visto.

Los **modelos basados en árboles de decisión**, como **Random Forest, Gradient Boosting y XGBoost**, fueron seleccionados como núcleo y partida de este trabajo por varias razones técnicas y prácticas (Fernández-Delgado et al., 2014):

- **Robustez ante datos tabulares heterogéneos:** Estos modelos no precisan una suposición previa sobre la distribución de los datos y realizan un buen manejo de distintos tipos de variables, escalas y relaciones no lineales, lo cual es perfecto para conjuntos como el que se ha empleado, el cual incluye métricas de juego de distinto tipo, escalas y significados.
- **Interpretabilidad relativa:** Aunque no son tan simples como un árbol de decisiones individual, modelos como **Random Forest** permiten valorar la importancia de las variables (*feature importance*), lo que permite y posibilita el entendimiento de qué acciones o métricas del jugador se encuentran más asociadas a comportamientos de fraude.

- **Capacidad para gestionar desbalanceo de clases:** En tareas donde la clase de interés es sustancialmente menor (como los tramposos en este caso), modelos como **Random Forest y XGBoost** permiten la modificación de pesos (*class_weight, scale_pos_weight*) para lidiar con este desbalanceo, sin necesidad de cambiar proactivamente los datos seleccionados.
- **Buen rendimiento sin necesidad de mucha ingeniería de atributos:** Al existir particiones sucesivas del espacio de características, estos modelos tienden a reportar buenos y más que decentes resultados, aún sin las transformaciones complejas o combinaciones manuales de atributos que requieren modelos lineales o redes neuronales.
- **Escalabilidad y eficiencia: XGBoost**, en particular, ha demostrado ser extremadamente eficiente en lo que se refiere a la parte computacional, tanto en el tiempo de entrenamiento como en el de predicción, gracias a su paralelización y optimización interna, hecho que hace permitir limitarse a escoger esos algoritmos en este contexto, ya que son una opción muy interesante para problemas prácticos dado que hay muchos ejemplos (Chen, T., & Guestrin, C., 2016).

En conclusión, el uso de algoritmos de tipo **árbol** es especialmente adecuado en este caso, concluyendo en precisión, facilidad de uso, robustez al ruido y un grado adecuado de interpretabilidad. Esta opción se encuentra ampliamente respaldada por la literatura científica existente en clasificación de datos tabulares y también es coherente en su uso en contextos con competiciones prácticas como las que encontramos en *Kaggle*, donde los modelos de *boosting* como **XGBoost** o el **Gradient Boosting** son los que suelen predominar en tareas de esta naturaleza.

4.6. Conclusión del flujo

Como conclusión del capítulo, el proceso de desarrollo algorítmico que se ha lleva a cabo en este trabajo ha seguido una trayectoria clara y secuencial, propia de un pipeline de aprendizaje automático definido como tal. Primeramente, se implementó una fase de preprocesado de los datos que integra limpieza, estandarización y selección de características más relevantes; y, por otro lado, se encuentra el tratamiento del desbalanceo de clases usando la técnica SMOTE en una de las ramas y partes del experimento.

A continuación, se llevó a cabo el entrenamiento de tres modelos basados en árboles (**Random Forest, Gradient Boosting y XGBoost**) usando diferentes estrategias de optimización de hiperparámetros (**Búsqueda manual, Grid Search y Random Search**). La diversidad de estrategias permitió explorar el comportamiento de cada uno de los algoritmos de acuerdo con sus parámetros para así poder obtener unas métricas finales lo más equilibradas posible.

Finalmente, se han evaluado y comparado todos los modelos generados a través de las métricas sobre **Accuracy, Precision, Recall y F1-Score** para que permitiesen seleccionar aquellas configuraciones que ofrecen un rendimiento más robusto y equilibrado en la detección de trampas en videojuegos multijugador online.

Este recorrido general seguido (**preprocesado → entrenamiento → evaluación comparativa**) no sólo permitieron maximizar la efectividad del modelo final sino también obtener una visión muy crítica y analítica del efecto que tienen distintas decisiones de diseño sobre el rendimiento global del sistema.

5. Descripción informática

En este capítulo se detalla la implementación técnica del sistema desarrollado para detectar trampas en videojuegos multijugador online y se exponen los pasos efectivos observados desde el punto de vista computacional, centrándose en la definición y la construcción de una arquitectura a base de módulos aplicando **Python**, además de emplear librerías para ciencia de datos y algoritmos para *Machine Learning*.

Se verá cómo se ha estructurado la información, el espacio de trabajo, las herramientas de cálculo utilizadas y el pipeline completo; es decir, desde la carga y el preprocesamiento del conjunto de datos inicial hasta la evaluación final de los modelos. Esta descripción permite ver cómo se ha traducido y formalizado el algoritmo propuesto para el planteamiento del problema en una solución informática que es escalable y fácil de reproducir.

5.1. Preparación del entorno de trabajo

La creación del entorno de trabajo es una parte muy importante en cualquier proyecto de ciencia de datos o *Machine Learning* porque permite asegurar que todas las herramientas, librerías y configuraciones necesarias están bien instaladas y gestionadas y, por lo tanto, hace garantizar una ejecución efectiva y reproducible del código.

En cuanto al desarrollo técnico en este Trabajo Fin de Grado, como ya se ha comentado, se ha realizado en **Python** como lenguaje de programación por la simplicidad de su sintaxis, por la gran comunidad de usuarios que tiene y también por su muy extenso ecosistema de librerías que están orientadas al análisis de los datos, a la visualización de los datos y a la sinergia que mantiene con el *Machine Learning*. La versión utilizada fue **Python 3.9** porque tiene compatibilidad con las principales librerías de las que se han necesitado hacer uso y porque se considera una versión muy estable.

Entorno de desarrollo

El desarrollo del proyecto se realizó en el entorno de desarrollo integrado **PyCharm**, provisto por **JetBrains**, que permite gestionar el código de forma ordenada, tiene una capacidad de depuración sencilla e intuitiva para código, dispone de soporte nativo para entornos virtuales y gestión del control de versiones.

Se configuró un entorno virtual aislado mediante *venv*, lo cual permite establecer dependencias sin tener efectos sobre el sistema global, además de permitir evitar conflictos entre versiones de librerías. De esta manera, ha permitido asegurar que el proyecto pueda ser fácilmente portable y reproducible en otras máquinas.

Las principales bibliotecas utilizadas a lo largo del desarrollo fueron:

- **pandas**: para la carga y manipulación de datos organizados en filas y columnas.

- **scikit-learn**: para el preprocesado de datos, modelado y evaluación.
- **imbalanced-learn**: para la aplicación de técnicas de balanceo como **SMOTE**.
- **xgboost**: para la implementación del algoritmo **XGBoost**.
- **matplotlib y seaborn**: para la visualización de datos.
- **openpyxl**: para exportación y manipulación de archivos *Excel*.
- **numpy**: apoyo en operaciones matemáticas y estructuras de datos numéricas.

Además, se hizo uso de funcionalidades propias del sistema operativo y del entorno **PyCharm** para organizar los scripts, automatizar la ejecución de procesos y gestionar los ficheros intermedios generados en cada fase del proyecto.

Esta configuración permitió disponer de un entorno robusto, flexible y bien adaptado al tipo de problema abordado, asegurando que el desarrollo pudiera realizarse de forma ordenada, reproducible y escalable.

5.2. Modelado

Durante la realización del trabajo, se optaron por seleccionar tres de los modelos de clasificación más populares y con un buen rendimiento en tareas con datos tabulares, los cuales son **Random Forest, Gradient Boosting y XGBoost**. A continuación, se presenta una breve descripción de cada uno de estos modelos, qué principales hiperparámetros se consideraron para el entrenamiento y evaluación y qué estrategias de búsqueda se usaron.

5.2.1. Modelos utilizados

- **Random Forest**: Este modelo de conjunto se basa principalmente en árboles de decisión. Es un modelo que crea muchos árboles durante el entrenamiento y cada árbol devuelve la clase que estima. Es un algoritmo robusto frente al sobreajuste que obtiene buenos resultados sin demasiada necesidad de ajuste y combinaciones. (Breiman, 2001).
- **Gradient Boosting:** En este caso se trata de un modelo de *boosting* secuencial donde cada nuevo árbol intenta corregir los errores de los árboles existentes. Este modelo puede sobreajustar (*overfitting*) con facilidad si sus hiperparámetros no se regulan, pero por el contrario se puede conseguir mucha efectividad con él (Friedman, 2001).

- **XGBoost:** Es una versión optimizada del algoritmo de **Gradient Boosting**, el cual está optimizado para ser muy eficiente, flexible y particularmente efectivo. Además, se considera un algoritmo de *boosting* regularizado porque también implementa regularización L1 y L2, y también trata explícitamente los valores perdidos, soporta la paralelización, lo que lo hace un algoritmo muy popular, especialmente en todo tipo de ámbitos de *Machine Learning* (Chen & Guestrin, 2016).

5.2.2. Hiperparámetros seleccionados

Para cada uno de los modelos, se definieron *grids* o combinaciones de búsqueda con los hiperparámetros más relevantes. Entre los principales parámetros establecidos se incluyen:

- **Random Forest**:
    - **n_estimators:** número de árboles en el bosque.
    - **max_depth:** profundidad máxima de cada árbol.
    - **min_samples_split y min_samples_leaf:** criterios mínimos para dividir nodos o establecer hojas.
    - **max_features:** número de características consideradas en cada división.
    - **class_weight:** para compensar el desbalanceo de clases.
- **Gradient Boosting**:
    - **n_estimators, learning_rate, max_depth:** principales controladores de complejidad.
    - **subsample:** proporción de muestras utilizadas en cada iteración.
    - **loss:** función de pérdida (por ejemplo, *"exponential"* o *"log_loss"*).
- **XGBoost**:
    - **n_estimators, learning_rate, max_depth:** parámetros clásicos.
    - **gamma:** reducción mínima de pérdida requerida para realizar una partición.
    - **reg_alpha, reg_lambda:** regularización L1 y L2.
    - **booster:** tipo de potenciador, como *"gbtree"*.
    - **scale_pos_weight:** utilizado para tratar el desbalanceo de clases.

(Scikit-learn developers, n.d.).

5.2.3. Estrategias de búsqueda de hiperparámetros

En la búsqueda por ajustar y encontrar las mejores combinaciones de hiperparámetros se utilizaron las siguientes metodologías:

- **Grid Search:** Incluye un conjunto delimitado de combinaciones y es exhaustivo. Resulta útil cuando hay tiempo y recursos de computación, aunque el coste es exponencialmente mayor en cuanto al número de parámetros y valores ya que la cantidad de combinaciones e iteraciones ejecutadas puede ser muy elevado dependiendo de los hiperparámetros esta.
- **Random Search:** Incluye una cantidad determinada de combinaciones elegidas aleatoriamente del *grid*. Durante el ajuste de hiperparámetros resulta menos exhaustivo que *Grid Search*, pero puede ofrecer buenas configuraciones de forma más rápida y eficiente. El número de combinaciones se establece con el número de iteraciones que se elijan.
- **Búsqueda manual:** Incluye combinaciones de hiperparámetros personalizadas en función de la experiencia previa, de pruebas empíricas y de la observación de resultados intermedios, lo que permitió centrar la búsqueda en determinados rangos de hiperparámetros que fueron observados a tener un mejor comportamiento.

(Koehrsen, 2018)

En cuanto a la información de cada modelo, también se ajustaron los hiperparámetros de los modelos utilizando las tres metodologías tanto en datos originales como en datos balanceados a partir de SMOTE, lo que permitió obtener un análisis más profundo del rendimiento de cada uno.

5.3. Evaluación y selección de métricas

La evaluación y selección de métricas es una fase crítica en cualquier proyecto de aprendizaje automático, ya que permite medir el rendimiento de los algoritmos implementados y tomar decisiones informadas sobre cuál es el modelo más adecuado para el problema planteado. En este trabajo, cuyo objetivo es detectar trampas en videojuegos multijugador online, este aspecto cobra especial importancia debido a la naturaleza **desbalanceada** del conjunto de datos, donde la clase minoritaria (jugadores no legítimos) representa un pequeño porcentaje del total de jugadores.

En este contexto, el uso de métricas estándar como el *accuracy* (exactitud) puede llevar a interpretaciones erróneas del rendimiento del modelo, dado que un clasificador trivial que prediga siempre la clase mayoritaria puede obtener una puntuación aparentemente alta sin aportar ninguna utilidad real. Por esta razón, aparte de esta, se seleccionó un conjunto de métricas más completas y adecuadas para evaluar clasificadores en escenarios de desbalanceo y no depender solo de la mencionada.

5.3.1. Métricas utilizadas

A continuación, se describen las métricas empleadas y su relevancia en el problema tratado (Saito & Rehmsmeier, 2015) (Powers, 2011):

- **Accuracy (exactitud)**

La **exactitud** representa el porcentaje total de predicciones correctas sobre el total de instancias evaluadas. Se define como:

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

Donde:

- **TP** (*True Positives*): trampas correctamente detectadas.
- **TN** (*True Negatives*): jugadores legítimos correctamente identificados.
- **FP** (*False Positives*): jugadores legítimos clasificados como tramposos.
- **FN** (*False Negatives*): trampas no detectadas.

Aunque es una métrica comúnmente utilizada, no es adecuada para tener como principal baza en *datasets* desbalanceados. En este proyecto, donde los jugadores tramposos son minoría, un modelo podría obtener un *accuracy* del 90% simplemente prediciendo que ningún jugador hace trampas, lo cual es inaceptable.

- **Precision (precisión)**

La **precisión** mide la proporción de predicciones positivas que son realmente correctas, es decir, qué tan confiable es el modelo al detectar trampas:

$$\text{Precision} = \frac{TP}{TP+FP}$$

En el ámbito de videojuegos, una precisión baja implica que el modelo está generando muchos **falsos positivos**, lo cual podría traducirse en que jugadores inocentes sean sancionados, lo que tiene graves implicaciones en la experiencia de usuario.

● **Recall (sensibilidad o cobertura)**

El **recall** representa la proporción de verdaderos positivos que el modelo es capaz de identificar, es decir, cuántos tramposos detecta realmente:

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$$

Un valor bajo de **recall** significa que el modelo está dejando pasar muchas trampas sin detectar (**falsos negativos**), lo cual compromete la seguridad del juego. En este tipo de problemas, mejorar el **recall** es fundamental, especialmente si se prioriza detectar el mayor número posible de infractores.

● **F1-Score (media armónica de precisión y recall)**

Dado que tanto la precisión como el *recall* son importantes y pueden estar en conflicto, se utiliza el **F1-Score** como métrica compuesta para equilibrar ambos aspectos:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}}$$

El **F1-Score** es particularmente útil en situaciones de desbalanceo como esta, ya que penaliza modelos que tengan un rendimiento descompensado entre precisión y *recall*. Por ejemplo, un modelo con precisión 0.9 y *recall* 0.1 tendrá un F1-Score bajo, reflejando una mala detección de trampas. Pese a que la precisión individual pueda parecer buena, se necesita un equilibrio entre las dos métricas.

5.3.2. Justificación de la elección de métricas

En conclusión y como resumen explicativo, la elección de las métricas para evaluar los modelos de clasificación no puede ser arbitraria, especialmente en contextos donde el desbalanceo de clases es significativo, como es el caso de este proyecto. En el problema de la detección de

trampas en videojuegos multijugador, como se ha visto, el número de jugadores tramposos es notablemente inferior al de jugadores legítimos, lo que convierte al *dataset* en uno claramente desbalanceado. En este escenario, métricas tradicionales como la exactitud (*accuracy*) pueden inducir a conclusiones erróneas si se interpretan de forma aislada.

Accuracy: limitaciones en entornos desbalanceados

La exactitud mide el porcentaje total de aciertos del modelo. Aunque es una métrica intuitiva y ampliamente utilizada, su utilidad se ve comprometida cuando una clase domina el conjunto de datos. Por tanto, aunque se ha incluido en el análisis para mantener una visión global del rendimiento, no se ha utilizado como criterio principal de selección.

Precision y Recall: un binomio clave

Dado que el objetivo del modelo es detectar jugadores que hacen trampas (la clase minoritaria), resulta crucial analizar con más detalle dos métricas complementarias:

- **Precision (Precisión):** Mide cuántos de los jugadores que el modelo identificó como tramposos realmente lo eran. Es esencial en contextos donde las falsas acusaciones pueden tener consecuencias graves (por ejemplo, suspender cuentas legítimas).
- **Recall (Exhaustividad o Sensibilidad):** Evalúa cuántos de los verdaderos tramposos fueron detectados correctamente por el modelo. Es vital para maximizar la capacidad de identificación del sistema anti-trampas, evitando que muchos tramposos pasen desapercibidos.

Estas métricas son especialmente importantes en sistemas de seguridad automatizados. Un sistema con alta precisión, pero bajo *recall*, por ejemplo, podría evitar penalizar a jugadores honestos, pero dejaría escapar a la mayoría de los tramposos. Por el contrario, un sistema con alto *recall* pero baja precisión detectaría a la mayoría de los tramposos, pero a costa de generar muchos falsos positivos. El objetivo, por tanto, es alcanzar un equilibrio entre ambas.

F1-Score: equilibrio entre precisión y recall

La métrica F1-Score, que es la media armónica entre precisión y *recall*, ofrece una visión más equilibrada en contextos desbalanceados. Su valor solo será alto si ambas métricas lo son, penalizando los casos en los que una de ellas es significativamente inferior. Por esta razón, se adoptó como la métrica principal para comparar modelos en este trabajo, dado que representa de forma más fiel la capacidad real de los clasificadores para identificar tramposos sin sobre reaccionar ante los falsos positivos.

¿Por qué no AUC-ROC?

Aunque el AUC-ROC es una métrica comúnmente utilizada, en este ámbito y campo de trabajo, para evaluar la capacidad discriminativa de modelos binarios, su utilidad disminuye y escasea en *datasets* muy desbalanceados. En esos casos, la curva *Precision-Recall* resulta más informativa (Saito & Rehmsmeier, 2015). No obstante, dado que el presente trabajo se enfocó

en métricas más interpretables para un entorno aplicado, y que los modelos se evaluaban principalmente sobre umbrales fijos (p. ej. 0.5 o 0.3), no se hizo uso extensivo de AUC-ROC en los resultados.

6. Resultados

Tras haber entrenado los modelos a partir de las distintas configuraciones y estrategias de búsqueda de hiperparámetros, se procedió a evaluar su rendimiento, usando diferentes conjuntos. Para ello se consideraron dos tipos de análisis: tanto el set original como aplicando la técnica del balanceo **SMOTE**. Para cada uno de estos casos, se examinaron distintas métricas que ofrecen una perspectiva completa del comportamiento de los clasificadores con y sin balanceo.

6.1. Métricas de evaluación utilizadas

Para la evaluación se emplearon las métricas más relevantes en clasificación binaria:

- **Accuracy (Exactitud):** proporción de instancias correctamente clasificadas.
- **Precision (Precisión):** proporción de verdaderos positivos entre las predicciones positivas.
- **Recall (Sensibilidad o tasa de verdaderos positivos):** proporción de positivos correctamente identificados sobre el total real de positivos.
- **F1-Score:** media armónica entre precisión y *recall*; útil cuando hay desbalanceo.

***Tiempo de entrenamiento:** tiempo necesario para entrenar el modelo con una configuración dada. A pesar de haberlo incluido en las tablas, no ha supuesto una métrica de evaluación a tener en cuenta, solo como guía de tiempos empleados e información extra.

6.2. Comparación de resultados sin SMOTE

Para comprobar el rendimiento de los modelos sin técnicas de balanceo como SMOTE se procedió con búsquedas de hiperparámetros de tres formas diferentes, en las cuales se encuentran: **Búsqueda manual**, **GridSearch** y **RandomizedSearch**. A continuación, se muestran los mejores resultados obtenidos por cada modelo de acuerdo con la métrica **F1-Score**, que permite captar el equilibrio entre **precisión** y **recall**.

Model	Accuracy	Precision	Recall	F1-Score	Train time (s)	Search
RandomForest	0.9222	0.7188	0.371	0.4894	1.09	ManualSearch
RandomForest	0.9222042	0.71875	0.3709677	0.48936		GridSearch
RandomForest	**0.9125**	**0.5909**	**0.4194**	**0.4906**	**0.71**	**RandomSearch**
GradientBoosting	0.9182	0.6533	0.3952	0.4925	30.92	ManualSearch
GradientBoosting	0.9157212	0.6315789	0.3870967	0.48		GridSearch
GradientBoosting	**0.9125**	**0.587**	**0.4355**	**0.5**	**26.14**	**RandomSearch**
XGBBoost	**0.8825**	**0.4323**	**0.5403**	**0.4803**	**2.19**	**ManualSearch**
XGBoost	0.8824959	0.4322580	0.5403225	0.48028		GridSearch
XGBoost	0.8865	0.4437	0.5081	0.4737	2.39	RandomSearch

Figura 1. Comparativas de modelos de clasificación según F1-Score sin SMOTE

Análisis de resultados sin SMOTE

Como se puede observar, **Gradient Boosting** alcanzó el mejor desempeño general, con un F1-Score de **0.5**, aunque su *recall* es relativamente bajo (0.4355). Esto indica que el modelo es capaz de identificar bien los casos positivos que predice, pero omite una cantidad significativa de ellos.

RandomForest y **XGBoost** presentan comportamientos bastante similares: precisión moderada y valores bajos de *recall*, lo que deriva en F1-Scores inferiores a 0.5. Se puede observar entonces que, sin una corrección del desbalanceo, los modelos tienden a favorecer la clase mayoritaria (no tramposos), perdiendo sensibilidad frente a la clase minoritaria (tramposos).

La diferencia entre *precision* y *recall* en todos los modelos es una evidencia clara de que el *dataset* original está desbalanceado. Aunque el *accuracy* supera el 90% en todos los casos, este valor es engañoso dada la distribución desigual entre clases.

6.3. Comparación de resultados con SMOTE

Del mismo modo que se hizo sin SMOTE, se comprobó el rendimiento de los modelos, pero con el uso de balanceo. Se realizaron búsquedas de hiperparámetros con los mismos buscadores que en el apartado anterior: **Búsqueda manual**, **GridSearch** y **RandomizedSearch.** De la misma forma, se procede a mostrar los mejores resultados obtenidos por modelo teniendo en cuenta la métrica **F1-Score**.

Model	Accuracy	Precision	Recall	F1-Score	Train time (s)	Search
RandomForest	0.901	0.8787	0.9304	0.9038	15.46	ManualSearch
RandomForest	0.902	0.8795	0.9318	0.9049	9.37	GridSearch
RandomForest	**0.9047**	**0.886**	**0.9291**	**0.907**	**24.8**	**RandomSearch**
GradientBoosting	0.9324	0.9366	0.9277	0.9321	38.61	ManualSearch
GradientBoosting	**0.9419**	**0.9366**	**0.948**	**0.9422**	**39.54**	**GridSearch**
GradientBoosting	0.9345	0.935	0.9338	0.9344	48.63	RandomSearch
XGBBoost	**0.9155**	**0.8947**	**0.9419**	**0.9177**	**2.22**	**ManualSearch**
XGBoost	0.9159	0.9073	0.9264	0.9168	1.06	GridSearch
XGBoost	0.9145	0.9007	0.9318	0.916	0.79	RandomSearch

Figura 2. Comparativas de modelos de clasificación según F1-Score con SMOTE

En este caso, el modelo **Gradient Boosting** da a comprender una vez más ser el mejor, alcanzó un F1-Score de hasta **0.9422** con un *recall* de **0.948** y una precisión de **0.9366**, lo que significa una mejoría de entorno al 90% en F1 respecto a su mejor configuración **sin SMOTE** lo que es altamente notable debido a que se ha mantenido también una precisión alta, es decir, el modelo presenta una buena habilidad para detectar trampas sin aumentar excesivamente con falsos positivos.

Random Forest también mostró una mejora muy notable. Su mejor modelo **con SMOTE** alcanzó un *recall* de 0.9291 y un *F1-Score* de 0.907, frente al 0.4096 observado **sin SMOTE**. Esto implica un incremento de aproximadamente un 85% en F1, y de entorno al 100% en **recall**,

confirmando que la técnica de balanceo fue crucial para mejorar su sensibilidad ante la clase minoritaria.

En el caso de **XGBoost**, se obtuvieron mejoras considerables en *recall* (hasta **0.9419**) y en *F1-Score* (hasta **0.9177**), aunque algo por debajo de los valores logrados por **Gradient Boosting**. Aun así, la precisión se mantuvo en niveles altos (~0.89–0.91), lo que hace de **XGBoost** una alternativa potente y bastante bien balanceada.

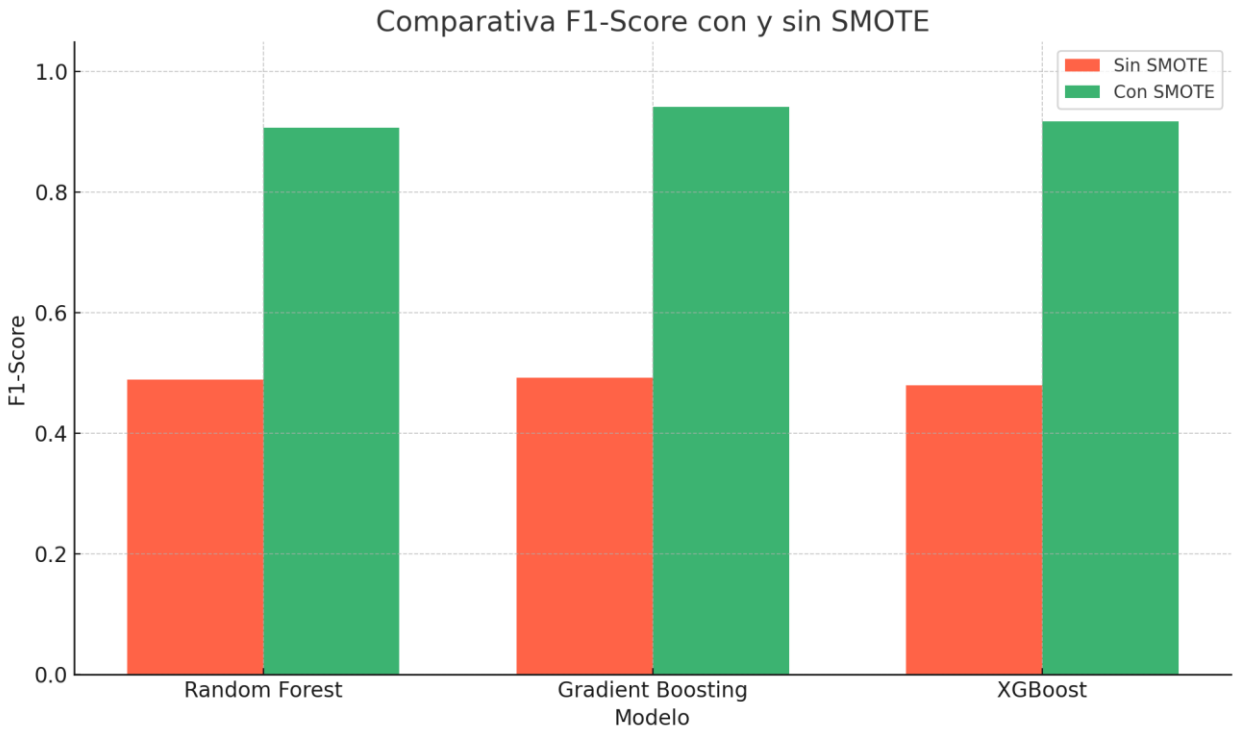

Figura 3. Gráfico comparativo con y sin SMOTE

Como se puede observar, los tres modelos obtuvieron un comportamiento mucho más equilibrado en el contexto de **SMOTE**. Es decir, la diferencia que se observa entre los valores de **precision** y **recall** se ha reducido considerablemente, siendo un reflejo de una mayor capacidad y suficiencia de los clasificadores en la detección de los tramposos sin una disminución en la precisión. También es el caso del **accuracy** o exactitud que, si ya era alto **sin SMOTE**, se ha mantenido un valor de **accuracy** parecido o en todo caso ligeramente más alto, aportando el hecho de que el balanceo no ha hecho disminuir el rendimiento general del modelo, sino que lo ha complementado y ha logrado equilibrar todo el modelo para mantener unos valores altos en todas sus características.

En cuanto al tiempo de entrenamiento, no se ha comentado en profundidad en el apartado anterior y en este debido a que, si bien puede ser algo significativo según qué datos se quieran extraer, depende mucho de otros factores que no son tan propios de los modelos o clasificadores. Como objeción, se observa que en algún caso la diferencia de tiempos es notable y en otros es bastante similar. Pero en definitiva como se ha comentado, entran en juego otros factores diferenciales.

Por último, estos resultados son un indicativo de que el preprocesamiento **con SMOTE** es necesario para mejorar la generalización del modelo en escenarios con un elevado desbalance de clases, lo cual es habitual y asiduo en la detección de trampas en los videojuegos multijugador online.

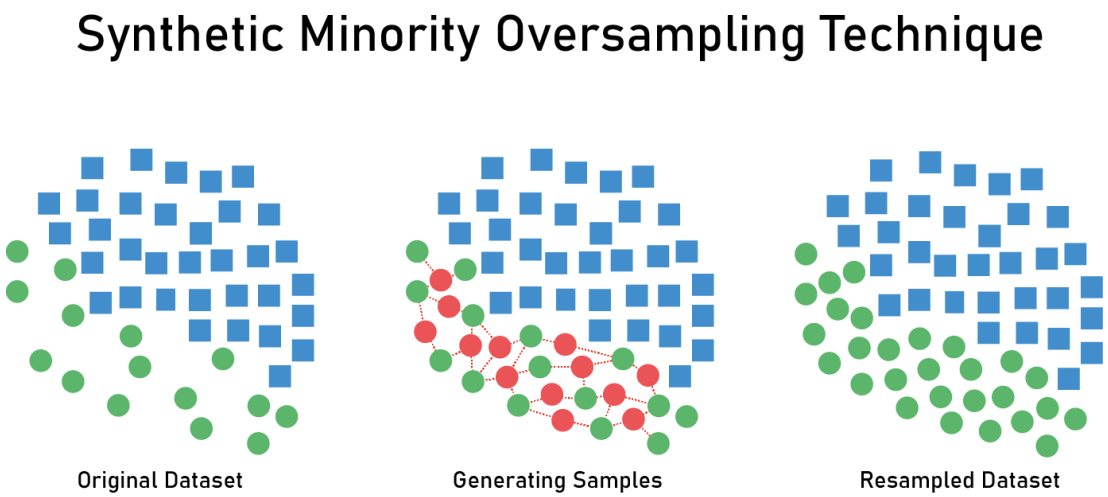

Figura 4. Explicación gráfica de SMOTE

Fuente: Jefferson, 2021

6.4. Comparación entre estrategias de búsqueda de hiperparámetros

Una parte fundamental del desarrollo de la actual tarea ha sido la comparación de las distintas estrategias para la búsqueda de hiperparámetros: **Grid Search, Random Search y Búsqueda Manual**. Durante todo el proceso, ha sido mostrado como cada una de las opciones influye en el desempeño de los modelos **sin SMOTE** y **con SMOTE**.

- **Random Forest**
  - **Sin SMOTE**: Las mejores configuraciones provinieron principalmente de la búsqueda **Random**, alcanzando un *F1-Score* de 0.4906. **Grid Search y Manual Search** ofrecieron resultados similares, pero no superiores.
  - **Con SMOTE**: La búsqueda **aleatoria** (**Random Search**) logró la mejor configuración con un *F1-Score* de 0.907, ligeramente superior al obtenido por **Grid Search** (0.9049) y la búsqueda manual (0.9038).
- **Gradient Boosting**
  - **Sin SMOTE**: **Grid Search y búsqueda manual** obtuvieron resultados casi idénticos (~0.49 F1). La búsqueda aleatoria ofreció mejoras puntuales, pero no significativas.

- **Con SMOTE**: **Grid Search** permitió alcanzar el mejor resultado global (F1 = 0.9422), seguido de **Random Search** (0.9344) y luego la **búsqueda manual** (0.9321). En este caso, la exploración de las distintas combinaciones de hiperparámetros resultó muy beneficiosa.

- **XGBoost**
  - **Sin SMOTE**: Todas las búsquedas arrojaron resultados parecidos, aunque **Grid Search** y manual obtuvieron un leve F1-Score superior (~0.4803 frente a 0.4737 en **Random Search**).
  - **Con SMOTE**: **Grid Search** logró un F1 de 0.9168, **Random Search** ~0.9160 y **búsqueda manual** 0.9177, por lo que los tres enfoques mostraron un comportamiento bastante parejo.

Análisis global:

- **Grid Search** permite explorar bien la totalidad del espacio de búsqueda, lo que puede llevar a los mejores resultados en algunos modelos, y hay que tener en cuenta que obtiene los mejores resultados en **Gradient Boosting con SMOTE**. Lo que tiene como inconveniente el uso de una gran cantidad de tiempo de cómputo y ejecutado.
- **Random Search** ofrece una buena relación rendimiento/tiempo, además de ser muy competitivo e incluso mejor en algunos casos concretos (ejemplo: **Random Forest con SMOTE**).
- **Búsqueda manual** puede ir bien para ser una primera aproximación o en caso de que se tenga conocimiento previo de alguna configuración, aunque en la mayoría de los casos será útil sin ser la más eficaz en casos más complejos (ejemplo: **XGBoost**).

6.5. Análisis de overfitting

El sobreajuste (*overfitting*) se produce cuando un modelo captura con demasiada precisión los datos de entrenamiento, extrayendo ruido o patrones que no se generalizan. Así que permitir que el modelo aprenda de manera óptima hace que el modelo que se entrene también tenga un buen resultado en la prueba de validación (test). La forma de evitar el sobreajuste es darse cuenta de que, a pesar de que el modelo es capaz de hacer predicciones para los datos de entrenamiento, también necesita tener resultados igualmente buenos (entradas y salidas) cuando se prueba con los datos de la prueba de validación (test). Es crucial tener indicios de sobreajuste para contar con un modelo que no solamente sea muy preciso para los datos de entrenamiento, sino que también lo sea para los de producción (IBM, n.d.).

Comparación entre validación y test

Si bien el objetivo principal era orientado al proceso de entrenamiento, para este análisis, también se compararon las métricas y resultados obtenidos durante el entrenamiento en contrapartida a los resultados reales obtenidos en el conjunto final de test, para así poder saber en que podían cambiar los valores entre los distintos procesos.

Model	F1-Score (Train)	F1-Scoe (Test)
RandomForest	0.9070	0.4580
GradientBoosting	0.9422	0.3599
XGBoost	0.9177	0.4579

Figura 5. Comparativas F1-Score entre train y test

Tal y como se puede observar perfectamente, hay una clara diferencia y caída entre los resultados y métricas al pasar de entrenamiento a test. Esto es una clara indicación de sobreajuste en todos los modelos, incluso habiendo utilizado varios tipos de técnicas durante el proceso.

Causas probables

- **Desbalance de clases residual**: A pesar de que **SMOTE** fue aplicado durante entrenamiento y validación, el *dataset* no fue balanceado artificialmente. Esto hace que exista una diferencia a nivel estructural entre los datos de entrenamiento y los de test que se poseen.

- **Complejidad de los modelos**: Modelos utilizados tales como **Gradient Boosting** con profundidades elevadas o muchos estimadores pueden obtener patrones demasiado específicos del conjunto de validación.

- **Thresholds por defecto**: Algunos resultados en test (por ejemplo, **Gradient Boosting con threshold**=0.5) daban *recall* muy bajo. Sin ajustar el umbral de decisión, los modelos tienden a ser conservadores con la clase minoritaria.

Ajuste del umbral

Se observó que al reducir el umbral (por ejemplo, de 0.5 a 0.3), el *recall* aumentaba sensiblemente, aunque de manera no significativa, aunque con una caída en precisión. Este ajuste es útil en contextos donde es preferible detectar más trampas (aunque haya algunos falsos positivos). Por ejemplo:

- *F1-Score* = 0.3599

- *Recall* = 0.2593
- *Precisión* = 0.5887
- *Accuracy* = 0.9027

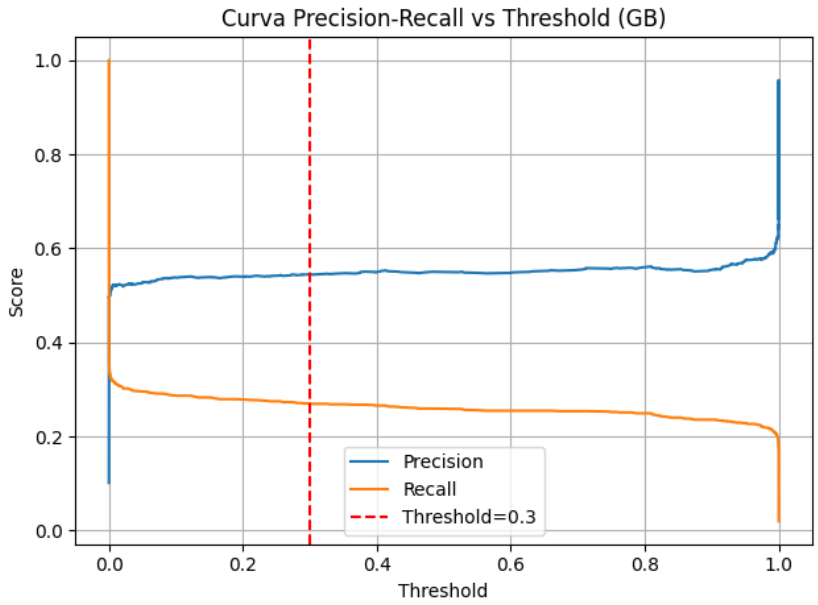

Figura 6. Curva comparativa GB

Como se ve en la figura, también se crearon curvas visuales para la comparación de resultados y métricas en la fase de test. Esto permite observar la caída y diferencia de resultados entre las distintas métricas utilizadas a diferencia de la fase de entrenamiento.

En el caso de **Random Forest** la cosa no sería muy diferente. Aplicando los mismos procesos y seleccionando la configuración de mejores parámetros obtenidos en el entrenamiento, se visualizaron resultados como:

- *F1-Score* = 0.4619
- *Recall* = 0.4577
- *Precisión* = 0. 4666
- *Accuracy* = 0.8896

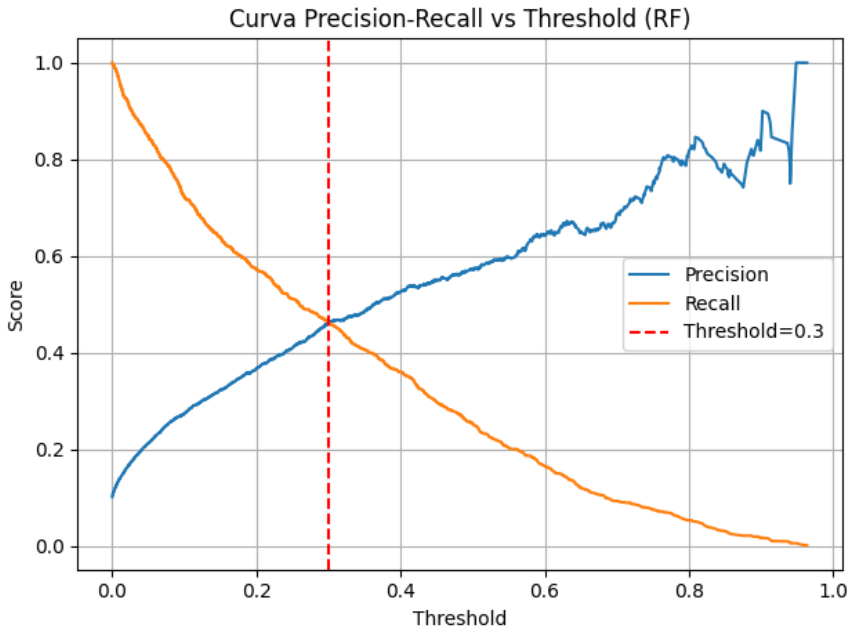

Figura 7. Curva comparativa RF

Misma situación para el clasificador **XGBoost**. Aplicamos mismos procesos y sus mejores configuraciones de parámetros y obtenemos:

- *F1-Score* = 0.3949
- *Recall* = 0.3394
- *Precisión* = 0.4722
- *Accuracy* = 0.8948

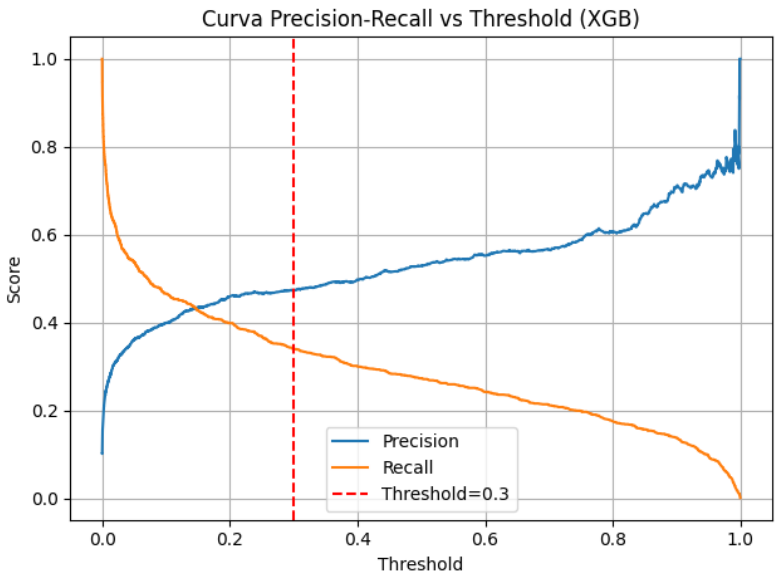

Figura 8. Curva comparativa XGB

Tal y como se vio en el caso de **Gradient Boosting**, en **Random Forest** y **XGBoost**, las diferencias entre resultados crecieron. En contrapartida a la fase de entrenamiento, los resultados presentados son más irregulares y hacen asumir comportamientos bastante distanciados.

Aunque la técnica de balancear utilizando **SMOTE** mejoró notablemente el rendimiento en validación, las pruebas muestran una cierta falta de generalización, sobre todo en *recall*.

Este sobreajuste puede explicarse a través de la distorsión de la distribución entre los datos de entrenamiento balanceados y el conjunto de test original.

6.6. Estrategias para mitigar el sobreajuste en futuras implementaciones

Como se ha comentado, el sobreajuste, u *overfitting*, es uno de los problemas más extendidos en el proceso de desarrollo de modelos de aprendizaje automático, y especialmente en condiciones de desbalance de clases en el mundo de la detección de trampas de videojuegos. En este trabajo se han aplicado técnicas como la validación cruzada, ajuste de hiperparámetros o aplicación de SMOTE para intentar mitigar y evitar este problema, pero las conclusiones extraídas una vez se ejecutaron las pruebas del último conjunto de test presentaron un claro menor rendimiento en los resultados obtenidos. Esto demuestra lo relevante e importante de aplicar ciertas medidas adicionales de refuerzo o una mayor generalización en futuros trabajos. A continuación, se van a ver algunas técnicas de interés para la reducción de este sobreajuste, basadas en diferentes artículos o literatura científica:

1. Validación cruzada anidada (Nested Cross-Validation)

La validación cruzada anidada es un método que se utiliza con el objetivo de estimar el error de generalización que tiene un modelo y sus hiperparámetros obtenidos o seleccionados. Es de particular importancia cuando se debe seleccionar entre varios modelos de aprendizaje o cuando se requiere la optimización de estos hiperparámetros.

Mientas que se utiliza la validación cruzada no anidada, tal y como se ha hecho, para ajustar los hiperparámetros y evaluar el rendimiento del modelo. La validación cruzada anidada se utiliza para estimar ese error de generalización que producen y se obtiene de los modelos y sus hiperparámetros. A nivel técnico, en un bucle interno se realiza una búsqueda en cuadrícula o **GridSearch** para encontrar los hiperparámetros y en un externo se evalúa directamente el rendimiento del modelo en el *dataset* de test.

En definitiva, esta técnica ofrece estimaciones más realistas al evitar el "filtro" de los datos de validación durante el ajuste (LabEx Technology Limited, n.d.).

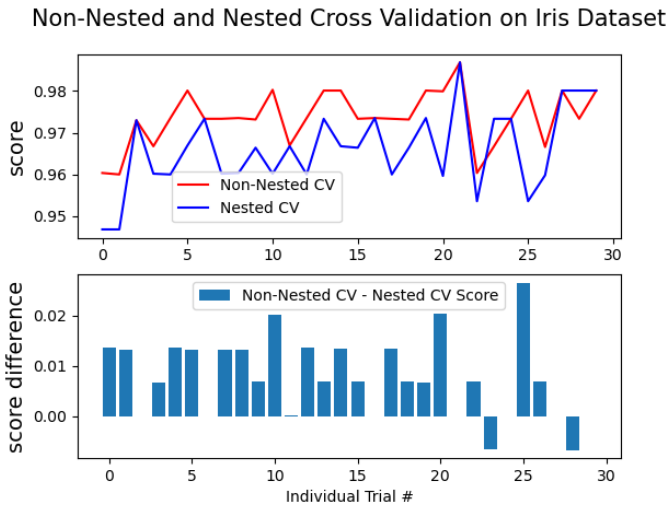

Figura 9. Comparativa de validación cruzada anidada y no anidada

Fuente: Documentación de scikit-learn

2. Regularización

La regularización es otra técnica utilizada en *Machine Learning* para evitar el sobreajuste (*overfitting*) de los modelos.

Dentro de las técnicas de regularización que se conocen, existen diferentes que se pueden aplicar, pero las más comunes y usadas son la regularización L1 (*Lasso*) y la regularización L2 (*Ridge*).

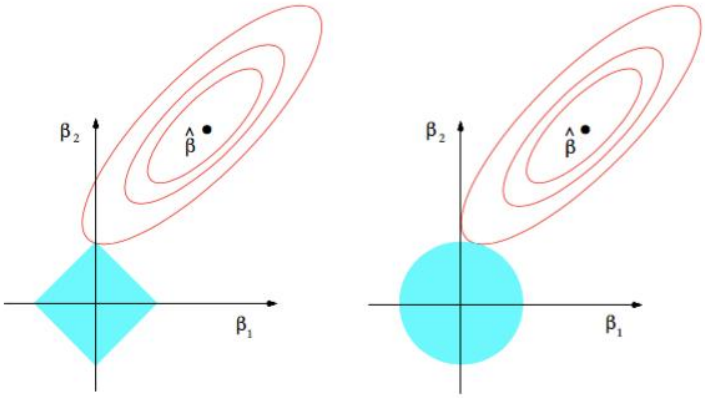

FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

Figura 10. Regularización L1 y L2

Fuente: Xiaoli, 2021

La **regularización L1** es capaz de añadir un término de penalización que es proporcional a la suma de los valores absolutos que tienen los coeficientes del modelo. Esto tiene como consecuencia forzar algunos coeficientes a cero, lo que puede ser bastante útil para hacer una clara selección de variables y así también poder eliminar las menos relevantes que se hayan obtenido. A nivel matemático, la función de coste con regularización L1 se define como:

$$J(w)=MSE(w)+\alpha\cdot||w||_1$$

Donde MSE(w) se define como la función de coste sin regularización, α es el parámetro de regularización y $||w||_1$ es la norma L1 de los coeficientes del modelo.

La **regularización L2** permite añadir un término de penalización que es proporcional a la suma de los cuadrados de los coeficientes del modelo. Esto permite y hace que se reduzcan los valores de los coeficientes, lo que puede ayudar a evitar el sobreajuste. La función de coste con regularización L2 se define como:

$$J(w)=MSE(w)+\alpha\cdot||w||^2_2$$

Donde MSE(w) es la función de coste sin regularización, α es el parámetro de regularización y $||w||_2$ es la norma L2 de los coeficientes del modelo.

A nivel técnico y usando **Sklearn** en **Python**, la regularización permite entrenar utilizando los mismos datos de entrenamiento:

```
lasso = Lasso(alpha=0.1)

lasso.fit(X_train, y_train)

ridge = Ridge(alpha=0.1)

ridge.fit(X_train, y_train)
```

Y evaluar el rendimiento que se obtiene de los modelos con el set de prueba y calculando el error cuadrático medio (MSE) ya comentado:

```
lasso_pred = lasso.predict(X_test)

lasso_mse = mean_squared_error(y_test, lasso_pred)

ridge_pred = ridge.predict(X_test)

ridge_mse = mean_squared_error(y_test, ridge_pred)

print("MSE Lasso:", lasso_mse)

print("MSE Ridge:", ridge_mse)
```

(Álvaro, 2023)

3. Eliminación de características recursivas

RFE (*Recursive Feature Elimination*) es un **método "wrapper" de selección de variables** que elimina iterativamente las características menos importantes según un modelo base (como regresión logística o árboles de decisión) y recalibra el modelo hasta quedarse con un número específico de variables. En cada iteración se genera un ranking de importancia, se descartan las menos importantes y se repite.

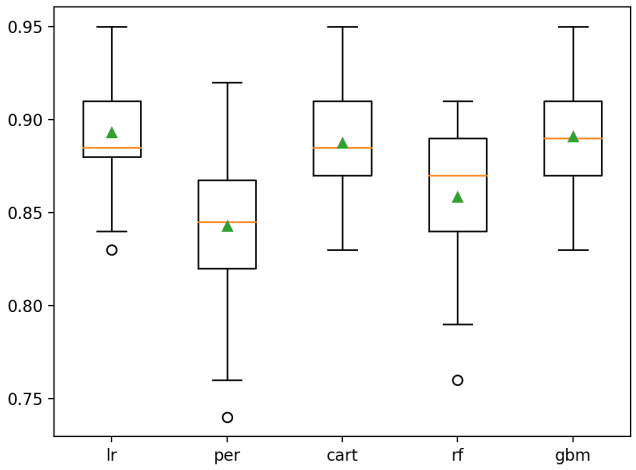

Figura 11. Comparativa de precisión entre clasificadores tras selección de características con RFE

Fuente: Brownlee, 2020

Detalladamente, el proceso que realiza paso a paso la eliminación de características recursivas es el siguiente:

1. Entrenamiento inicial

Se entrena un modelo (por ejemplo, una regresión logística o un árbol de decisión) usando **todas las características disponibles**.

2. Evaluación de importancia

El modelo genera un **ranking de importancia** de las características:

- En modelos lineales, esto puede derivarse de los coeficientes absolutos.
- En modelos basados en árboles, se calcula con métricas como la ganancia de información o Gini.

3. Eliminación de la menos importante

Se **elimina la característica menos importante** (o el número definido por el usuario), y se vuelve a entrenar el modelo con las variables restantes.

4. Iteración

Este proceso se **repite recursivamente**, eliminando en cada paso la menos útil, hasta que se alcanza el número deseado de características (*n_features_to_select*).

5. Resultado final

El algoritmo devuelve:

- Las características seleccionadas.
- Un *ranking* completo de todas las variables.
- Los conjuntos transformados con solo las variables seleccionadas (opcional).

A nivel técnico en **Python**, un ejemplo para el uso de esta técnica **RFE** en un contexto de clasificación podría ser:

```
# define dataset

X, y = make_classification(n_samples=1000, n_features=10, n_informative=5,
                           n_redundant=5, random_state=1)

# create pipeline

rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=5)

model = DecisionTreeClassifier()

pipeline = Pipeline(steps=[('s',rfe),('m',model)])

# evaluate model

cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

n_scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-
                           1, error_score='raise')
```

(Brownlee, 2020).

4. Técnicas de balanceo más avanzadas

Aunque **SMOTE** ha sido útil, métodos como **Borderline-SMOTE, SMOTEENN o ADASYN** ofrecen mejoras al generar datos sintéticos más relevantes o limpiar ejemplos mal etiquetados. Estas variantes pueden reducir el ruido y mejorar el rendimiento final del modelo. Más concretamente:

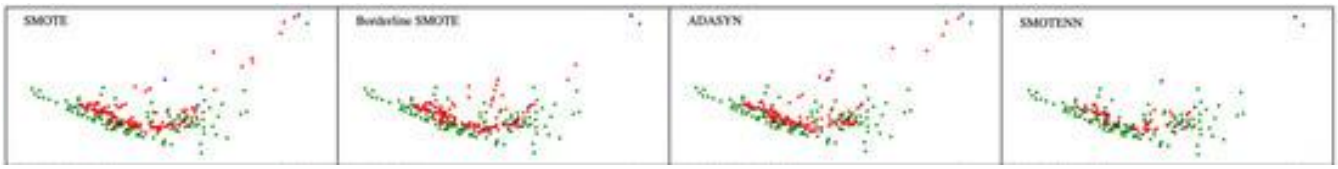

Figura 12.Comparativa de los cuatro métodos de balanceo aplicados a un conjunto de datos

Fuente: Lyu et al., 2025

1. ADASYN (Adaptive Synthetic Sampling)

Basado en **SMOTE**, pero **más selectivo**:

- Identifica las instancias minoritarias más cercanas a la frontera con la clase mayoritaria.
- Calcula un **índice de dificultad** local (relación mayoría/minoría).
- Genera más muestras sintéticas en regiones donde minoría es difícil de clasificar.

Beneficios:

- Refuerza las zonas problemáticas de decisión.
- Puede mejorar la discriminación donde es más necesario.

Riesgos: si hay ruido o *outliers* en zonas bajas de densidad, puede amplificarlos.

2. Borderline-SMOTE

Variante que **se centra en la frontera de decisión**:

- Reconoce instancias minoritarias cercanas a la clase mayoritaria (zonas limítrofes).
- Genera nuevos ejemplos **solo** cerca de estas zonas, interpolando entre vecinos similares.

Ventaja: fortalece el modelo donde suele fallar.

Riesgos: si los puntos limítrofes están distorsionados, se corre el riesgo de reforzar ruido.

(Goswami, 2020).

3. **SMOTE-ENN**

SMOTE-ENN es una técnica de balanceo de clases desarrollada por Batista et al. (2004) que combina dos enfoques:

- **SMOTE** (*Synthetic Minority Over-sampling Technique*): genera ejemplos sintéticos de la clase minoritaria como ya se ha aplicado anteriormente en este trabajo.

- **ENN** (*Edited Nearest Neighbors*): elimina observaciones conflictivas de ambas clases.

El objetivo es mejorar la calidad del conjunto de datos balanceado, aumentando la clase minoritaria y eliminando ruido o ambigüedad en la mayoría y minoría.

Funcionamiento de SMOTE-ENN

Parte 1: **SMOTE** (sobremuestreo sintético)

1. Se elige aleatoriamente una observación de la clase minoritaria.

2. Se calculan sus k vecinos más cercanos (por defecto, k=5).

3. Se genera una nueva muestra sintética interpolando entre la observación original y uno de sus vecinos:

$$\text{nuevo} = \text{original} + \lambda * (\text{vecino} - \text{original}) \text{ con } \lambda \in [0,1]$$

4. Se repite este proceso hasta alcanzar la cantidad deseada de ejemplos minoritarios.

Parte 2: **ENN** (depuración con vecinos cercanos)

1. Se define un valor de K para vecinos (por defecto K=3).

2. Para cada observación del *dataset* (de ambas clases), se hallan sus K vecinos más cercanos.

3. Si la clase de la observación no coincide con la clase mayoritaria de sus vecinos, se elimina esa observación y sus vecinos.

4. Se repite hasta estabilizar el *dataset*.

Ventajas:

- Genera datos sintéticos realistas para la clase minoritaria.
- Elimina instancias ruidosas o conflictivas tanto de la minoría como de la mayoría.

- Mejora la definición del espacio de decisión, especialmente en los bordes entre clases.
- Mejora la generalización del modelo en test al reducir *overfitting* provocado por ruido.

(Andhika, 2021)

Riesgos:

- Pérdida de datos útiles si están cerca de los límites de decisión entre clases o si el *dataset* es pequeño (Batista et al., 2004).
- Alta sensibilidad del parámetro k debido a poder generar datos irreales o eliminar en exceso (García et al., 2015).
- Coste computacional elevado ocasionado por cálculos de distancias para generar datos sintéticos y comparaciones de vecinos para cada instancia (Fernández et al., 2018).

Existen otros tipos de técnicas que pueden ayudar a controlar el sobreajuste e intentar mitigarlo de la mejor manera posible, como sería el caso del **aumento de datos (Data Augmentation).** La *data augmentation* consiste en un conjunto de técnicas que pueden incrementar la cantidad de datos por medio de la modificación de versiones de datos existentes(no confundir con la generación automática de datos totalmente artificiales como hemos visto anteriormente). Sin embargo, esta técnica está ciertamente orientada en procesamiento de imágenes, en áreas y sectores como detección e identificación de objetos, segmentación semántica, reconocimiento de acciones y gestos o en la clasificación de paquetes de imágenes (Murel & Kavlakoglu, 2024). Es por esto, que no se va a desarrollar más esta técnica en este trabajo ya que no se adecua completamente con el tipo de datos manejados.

7. Conclusiones y trabajo futuro

Durante todo este Trabajo de Fin de Grado se ha planteado un problema claro referente al ámbito del desarrollo de videojuegos: la detección automática de comportamientos tramposos en entornos multijugador online. La aplicación de técnicas de aprendizaje automático a este contexto comentado ha permitido construir, evaluar y comparar distintos modelos de clasificación capaces de identificar a jugadores que infringen las reglas del juego como una costumbre. En este último capítulo, se presentan las conclusiones más relevantes extraídas del desarrollo del proyecto, así como propuestas y recomendaciones para futuras implementaciones de mejora y ampliación del trabajo realizado.

7.1. Conclusiones

A lo largo de este Trabajo de Fin de Grado, se ha desarrollado y profundizado un sistema de clasificación orientado a la **detección automática de trampas en videojuegos multijugador online**, utilizando técnicas de aprendizaje automático sobre un conjunto de datos real y etiquetado.

Después de un análisis de exploración y una fase ardua de preprocesado, se evaluaron distintos modelos de clasificación: **Random Forest**, **Gradient Boosting** y **XGBoost**, a los cuales se les aplicaron técnicas de selección de características, validación cruzada y optimización de hiperparámetros mediante **Grid Search**, **Random Search** y **búsqueda manual**. Además, se compararon los resultados obtenidos **con y sin** la técnica de balanceo **SMOTE**, dada la marcada desproporción entre clases (tramposos y no tramposos).

En función de los objetivos planteados al inicio del trabajo, se puede decir que se ha logrado llegar a todos ellos. Se ha desarrollado un pipeline completo de **Machine Learning** que parte desde las etapas de preprocesamiento, balanceo de clases, selección de características, entrenamiento de modelos y evaluación mediante diferentes métricas. Además de esto, se ha realizado una comparativa entre diferentes algoritmos y diferentes parámetros de optimización de manera que se pueda valorar cuál se acomoda mejor al problema planteado. También se ha profundizado en el estudio del impacto del desbalanceo de clases en la calidad del modelo y se han valorado diferentes estrategias para mitigar este problema, lo que permite reforzar todavía más el carácter investigativo y aplicado del proyecto.

7.1.1. Evaluación técnica y hallazgos clave

Uno de los principales y mayores hallazgos de todo el trabajo es que el uso de técnicas de balanceo como **SMOTE** ha resultado de suma importancia para mejorar el rendimiento y el devenir del sistema de detección y del trabajo de investigación. En la versión original extraída, el *dataset* presentaba un fuerte desbalance de clases, con una gran mayoría de usuarios

correspondientes a jugadores legítimos o no tramposo. Este desequilibrio afectaba negativamente al desempeño de los modelos, que hacía tender a favorecer la clase mayoritaria y a ignorar la minoritaria, precisamente la que es más importante detectar en este contexto para poder identificar a esos usuarios tramposos no deseados.

En cuanto a los resultados obtenidos sin SMOTE, a pesar de mostrar una precisión aceptable y correcta, reflejaban niveles bajos de **recall y F1-Score**, dando a ver la dificultad de los modelos para capturar la clase de los tramposos. En contraposición, tras aplicar **SMOTE**, los valores de **recall y F1** se incrementaron notablemente, llegando incluso a duplicarse en algunos casos. Esto demuestra que la calidad del preprocesamiento de los datos es tan importante como el propio modelo elegido, y puede marcar una diferencia importante en el rendimiento final del sistema.

Refiriéndonos a los algoritmos, el modelo **Gradient Boosting** fue el que devolvió mejores resultados de forma general, tanto **con,** como **sin SMOTE**. En su configuración óptima, alcanzó un **F1-Score** superior a 0.94, reflejando una excelente capacidad para identificar tramposos sin un elevado número de falsos positivos. **Random Forest y XGBoost** también lograron rendimientos competitivos con resultados bastante parecidos, especialmente tras el ajuste de hiperparámetros y con datos balanceados.

7.1.2. Importancia de la métrica F1-Score y del umbral

A lo largo del desarrollo del proyecto se confirmó que, en contextos con clases desbalanceadas, el uso de métricas tradicionales como el **accuracy** puede resultar engañosas. Todos los modelos superaban el 90% en esta métrica incluso sin detectar apenas tramposos, lo que da pie a su limitación. Por tanto, se ha priorizado el uso de métricas más informativas como **precision**, **recall** y especialmente el **F1-Score**, que representa un equilibrio entre ambas.

Asimismo, se ha añadido un análisis del umbral de decisión del clasificador, demostrando que modificarlo puede cambiar ligeramente el comportamiento del modelo. Por ejemplo, al reducir el umbral a 0.3, se logró incrementar el **recall** (capacidad para detectar tramposos), sacrificando parte de la precisión. Esta estrategia es especialmente útil en aplicaciones prácticas, donde se puede preferir generar alertas o poner en vigilancia, con cierto margen de error, antes que dejar pasar comportamientos sospechosos.

7.1.3. Visión crítica y reflexiva del trabajo

Una de las aportaciones más relevantes e importantes de este TFG es el **enfoque sistemático y comparativo** aplicado al problema. No se limitó a probar un modelo, sino que se evaluaron múltiples clasificadores, distintas estrategias de búsqueda de hiperparámetros (manual, aleatoria y grid search), diferentes técnicas de preprocesamiento, y se llevó a cabo una evaluación final sobre un conjunto de test completamente aislado.

Este enfoque exhaustivo permitió detectar fenómenos como el **overfitting**, mayoritariamente en algunos modelos sin técnicas de regularización adecuadas. También se ha podido comprobar que el buen rendimiento en validación no garantiza un rendimiento similar en test, lo que refuerza la importancia de contar con datos representativos y bien particionados y no quedarse anclado en una única tanda de resultados de una fase.

Por otro lado, este proyecto pone de manifiesto una de las **limitaciones actuales en la detección automática de trampas**: los *datasets* disponibles, que no sean privados o confidenciales, aunque valiosos, pueden estar desactualizados o no captar la totalidad de comportamientos anómalos. En consecuencia, cualquier sistema real debería ser acompañado por mecanismos de aprendizaje continuo y adaptación.

7.1.4. Aplicabilidad en el contexto de videojuegos

Desde el punto de vista de su aplicación práctica, los resultados de este TFG son especialmente relevantes y aclaratorios para la industria de los videojuegos multijugador en un contexto académico. Algunos de los sistemas *anti-cheating* actuales, aunque eficaces, en muchas ocasiones se basan en reglas más rígidas o heurísticas, lo que los puede hacer ciertamente vulnerables y débiles a trampas más sofisticadas y expertas. En contrapartida, los modelos basados en aprendizaje automático tienen la capacidad de adaptarse a nuevos patrones de comportamiento, aprender y evolucionar con el tiempo.

Este trabajo podría servir como base para un sistema complementario de detección temprana, que funcione como filtro previo antes de una revisión humana, o que alimente sistemas de reputación o sanción automática. Además, el sistema puede integrarse en entornos reales mediante *APIs* o microservicios, gracias a su implementación modular en **Python**.

7.2. Trabajo futuro

A lo largo de este trabajo se ha evidenciado el potencial del aprendizaje automático para abordar la detección de trampas en videojuegos multijugador online. Sin embargo, este proyecto representa solo un primer paso dentro de una línea de investigación y desarrollo mucho más amplia y capaz. A continuación, se presentan diversas direcciones o rumbos en los que podría continuar o ampliarse este trabajo, tanto desde un punto de vista técnico como desde una perspectiva práctica.

7.2.1. Ampliación y mejora del conjunto de datos

Uno de los factores más determinantes en el rendimiento de los modelos ha sido la disponibilidad de datos etiquetados y representativos que se obtuvieron del *dataset*. En este sentido, una ampliación del conjunto de datos ya sea mediante la colaboración con estudios de videojuegos o el uso de técnicas de *web scraping* sobre foros y comunidades, podría mejorar significativamente la capacidad de mejorar, avanzar y evolucionar del sistema.

Además, podrían considerarse técnicas más sofisticadas de generación de datos sintéticos como se ha comentado, como pueden ser *data augmentation* con *autoencoders* o incluso el uso de modelos generativos adversarios (*GANs*) adaptados a tabular data (Xu et al., 2019), lo cual puede ser útil para enriquecer la clase minoritaria sin recurrir exclusivamente a la técnica **SMOTE**.

7.2.2. Exploración de modelos más avanzados

Aunque **Random Forest, Gradient Boosting y XGBoost** han dado muy buenos resultados, existen otros algoritmos más recientes y potentes que podrían ser evaluados y probados en esta misma tarea:

- **LightGBM**: una alternativa optimizada al **Gradient Boosting**, especialmente eficiente con grandes volúmenes de datos y que incorpora *histogram-based learning*.

- **CatBoost**: especialmente interesante cuando se trabaja con variables categóricas (que aquí fueron preprocesadas), ya que maneja automáticamente su codificación (Amat & Escobar, 2021).

- **Modelos neuronales ligeros**: como *MLPs* para datos tabulares o arquitecturas como *TabNet* (Arik & Pfister, 2021), que combinan atención con interpretabilidad.

7.2.3. Incorporación de variables de comportamiento en tiempo real

Actualmente, el *dataset* incluye estadísticas agregadas, pero en una aplicación real y en un entorno de test real sería muy positivo e interesante incorporar **datos temporales** o de comportamiento **en tiempo real**, como patrones de movimiento, precisión por minuto, velocidad de reacción o uso extraño del ratón/teclado. La monitorización de datos en tiempo real equivale a la recogida, al tratamiento y al análisis continuado de la información a medida que ésta se produce (Reportz et al., 2025). Estas variables podrían alimentar modelos secuenciales como **LSTM** o **transformers temporales**, y dar lugar a un sistema más predictivo que reactivo.

7.2.4. Evaluación en entornos productivos o simulados

Un paso crucial en el trabajo futuro podía ser la implementación e integración del modelo en un **entorno de prueba**, como un servidor de un videojuego con tráfico real o simulado. Esto permitiría analizar aspectos como:

- Latencia de predicción.
- Tasa de falsos positivos y su impacto en la experiencia del usuario.
- Comportamiento del sistema bajo ataques adaptativos.

Además, podría estudiarse la aceptación y opiniones del sistema por parte de la comunidad de jugadores, ya que su percepción sobre cómo de justo o adecuado sería el sistema *anti-cheating* es esencial.

7.2.5. Aplicación ética y explicabilidad del modelo

En ciertas situaciones reales sensibles como la detección de trampas, la **explicabilidad del modelo (model interpretability)** se vuelve de suma importancia. Herramientas como **SHAP** o **LIME** podrían integrarse para mostrar a los administradores o moderadores por qué un jugador fue clasificado como tramposo y que puedan ver qué comportamientos extraños se están observando.

Además, es importante considerar cuestiones éticas: ¿cómo se maneja un falso positivo? ¿Qué mecanismos de apelación, revisión o reclamación debería tener el sistema? Añadir estos aspectos garantizaría no solo la eficacia técnica del sistema, sino también su **aceptación social y legal** (Ribeiro, Singh & Guestrin, 2016).

En definitiva, este trabajo abre múltiples vías de exploración futura, tanto desde el punto de vista técnico como aplicado. La detección automática de trampas en videojuegos, y más en multijugadores online, es un problema complicado y cambiante, y la capacidad de los modelos para adaptarse, explicar sus decisiones y trabajar en condiciones reales marcará la diferencia entre una solución experimental y un sistema listo para producción que sea realmente eficaz y justo. El desarrollo de futuros sistemas híbridos, con módulos automáticos y ayuda de revisión humana, puede representar la siguiente generación de defensas y detección anti-trampas en videojuegos multijugador.

Referencias

- **Álvaro. (2023).** ***Regularización en Machine Learning. Ejemplo con Python*****. MachineLearningParaTodos.**
- **Amat, J. & Escobar, J. (2021).** ***Forecasting series temporales con gradient boosting: Skforecast, XGBoost, LightGBM y CatBoost.***
- **Andhika, R. (2021).** ***Imbalanced Classification in Python: SMOTE-ENN Method*****. Towards Data Science.**
- **Arik, S. Ö., & Pfister, T. (2021).** ***TabNet: Attentive Interpretable Tabular Learning.*** **Proceedings of the AAAI Conference on Artificial Intelligence, 35(8), 6679–6687.**
- **Batista et al. (2004).** ***A study of the behavior of several methods for balancing machine learning training data*****.**
- **Breiman, L. (2001).** ***Random Forests*****. Machine Learning, 45(1), 5–32.**
- **Brownlee, J. (s. f.).** ***Recursive Feature Elimination (RFE) for Feature Selection in Python*****. Machine Learning Mastery.**
- **Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002).** ***SMOTE: Synthetic Minority Over-sampling Technique*****. Journal of Artificial Intelligence Research, 16, 321–357.**
- **Chen, T., & Guestrin, C. (2016).** ***XGBoost: A Scalable Tree Boosting System*****. Proceedings of the 22nd ACM SIGKDD.**
- **García, S., Luengo, J., & Herrera, F. (2015).** ***Data preprocessing in data mining*****.**
- **Goswami, S. (2020).** ***Class imbalance: SMOTE, Borderline-SMOTE & ADASYN*****. Towards Data Science.**
- **Fernández, A., García, S., Galar, M., Prati, R. C., Krawczyk, B., & Herrera, F. (2018).** ***Learning from Imbalanced Data Sets*****. Springer.**
- **Fernández-Delgado, M., Cernadas, E., Barro, S., & Amorim, D. (2014).** ***Do we need hundreds of classifiers to solve real world classification problems? Journal of Machine Learning Research.***
- **Friedman, J. H. (2001).** ***Greedy Function Approximation: A Gradient Boosting Machine*****. The Annals of Statistics, 29(5), 1189–1232.**
- **IBM. (n.d.).** ***Overfitting*****. IBM.**
- **Kotkov, D. (2021).** ***Gaming Bot Detection: A Systematic Literature Review*****.**
- **LabEx Technology Limited. (n.d.).** ***Nested cross-validation for model selection*****. LabEx.**
- **Lyu, J., Yang, J., Su, Z., & Zhu, Z. (2025).** ***LD-SMOTE: A Novel Local Density Estimation-Based Oversampling Method for Imbalanced Datasets*****. Symmetry, 17(2), 160.**
- **Murel, J., & Kavlakoglu, E. (2024, 7 de mayo).** ***¿Qué es el aumento de datos?*** **IBM.**

- **Powers, D. M. (2011).** ***Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation*****. Journal of Machine Learning Technologies.**
- **Reportz, Basta, R. (2025, 29 abril).** ***Real-Time Data Tracking Explained: Strategies for 2025*****. Reportz.**
- **Ribeiro, M. T., Singh, S., & Guestrin, C. (2016).** ***"Why should I trust you?": Explaining the predictions of any classifier*****. ACM SIGKDD.**
- **Saito, T., & Rehmsmeier, M. (2015).** ***The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets*****.**
- **Scikit-learn developers. (n.d.).** ***sklearn.ensemble.*** **Scikit-learn.**
- **Scikit-learn developers. (n.d.).** ***Validación cruzada anidada y no anidada*****. Scikit-learn.**
- **Willman, M. (2020).** ***Machine Learning to identify cheaters in online games*****.**
- **William, K. (2018).** ***Intro to Model Tuning: Grid and Random Search*****.**
- **C, X. (2021, 6 febrero).** ***Intuitive and Visual Explanation on the differences between L1 and L2 regularization*****. LinkedIn.**
- **Xu, L., Skoularidou, M., Cuesta-Infante, A., & Veeramachaneni, K. (2019).** ***Modeling Tabular Data using Conditional GAN*****. NeurIPS.**
- **Zhang, J., Sun, C., Gu, Y., Zhang, Q., Lin, J., Du, X., & Qian, C. (2022).** ***Identify as a human does: A pathfinder of next-generation anti-cheat framework for first-person shooter games*****. University of Hong Kong**