Universitat Autònoma de Barcelona

Dipòsit digital de documents de la UAB

This is the **published version** of the bachelor thesis:

Herrera Palacio, Pablo; Sánchez Albaladejo, Gemma, dir. Ordenado Automático de Inventario según preferencias de usuario. 2022. (958 Enginyeria Informàtica)

This version is available at https://ddd.uab.cat/record/264199

under the terms of the  license

Ordenado Automático de Inventario según preferencias de usuario

Pablo Herrera Palacio

Resumen– PGM (PvP Game Manager) es un plugin de código abierto sobre el videojuego de estilo sandbox Minecraft [1]. Las modalidades de juego de PGM [2] consisten en mapas específicos en los que dos o mas equipos luchan por conseguir un objetivo. Para ello disponen de un kit con objetos útiles, donde acceder a los objetos de forma ágil es fundamental para ganar. Por ello los jugadores se reordenan los objetos del kit acorde a sus preferencias; sin embargo, este orden se perdía cada vez que el jugador moría. El objetivo de este trabajo surge a raíz de las peticiones de los usuarios de poder mantener su orden de preferencia de los kits, ya que que encuentran muy tedioso cambiar el orden de los objetos a menudo, y consiste en predecir automáticamente el orden preferido para cada jugador. Para ello se han recogido y analizado los datos de miles de partidas, visualizando y extrayendo conclusiones de los diferentes patrones de comportamiento de los usuarios. Finalmente se han diseñado y evaluado distintos modelos de predicción basados en *naive bayes* logrando reducir el error, entendido como cantidad de objetos que los usuarios cambian de posición en el inventario, en un factor de x2,26 en comparación con mantener el orden por defecto del videojuego.

Palabras clave– preferencias de usuario, machine learning, recolección de datos, minecraft, análisis de comportamiento

Abstract– PGM (PvP Game Manager) is an open source plugin for the sandbox videogame minecraft [1]. PGM [2] gamemodes consist of specific maps where two or more teams fight to accomplish an objective. To accomplish the goal players are given kits with useful items, and quickly accessing items is fundamental to win. For this reason players order their kits according to their preferences; however, order was lost every time the player died. The goal of this project is rooted in a demand from the playerbase of being able to keep their preferred order, because they find it tedious to change the order often, and consits in automatically predicting the preferred order. To accomplish that goal, data from thousands of matches will be gathered and analyzed, visualizing and extracting conclusions of different behavior patterns in users. Finally several *naive bayes* based prediction models have been designed and evaluated. The error, understood as the amount of items moved by players, has been lowered by a factor of x2.26 in relation to using default kits' order in the game.

Keywords– user preferences, machine learning, data gathering, minecraft, behavorial analysis

1 Introducción

El perfilado de usuarios y la personalización de contenidos *online* tiene una gran relevancia actualmente en diversidad de sectores, desde publicidad hasta recomendaciones en plataformas de vídeo bajo demanda [3]. Este es un problema complejo, muy en auge y que conlleva el estudio de comportamientos habituales de los usuarios.

Minecraft [1] es un videojuego del estilo *sandbox* que ofrece un mundo donde el jugador es libre para usar su creatividad [4], o poner a prueba su habilidad e ingenio. El juego en su modo normal consiste en la supervivencia, búsqueda de recursos y construcción, en modo individual o servidor, pero también es común ver servidores con *plug-ins* que modifican las mecánicas normales del juego e imponen reglas, mecánicas u objetivos distintos. Nuestro caso de uso se centra en unas modalidades de dicho juego ejecutadas por el *plug-in* de código abierto *PGM* (PvP Game Manager) [2], con el cual comparto una larga trayectoria de desarrollo.

Las modalidades de juego de *PGM* [2] consisten en mapas específicos creados para que dos o más equipos luchen

• E-mail de contacto: pablo.herrerap@autonoma.cat
• Mención realizada: Computación
• Trabajo tutorizado por: Gemma Sanchez Albaladejo (Ciencias de la Computación)
• Curso 2021/2022

por conseguir un objetivo, por ejemplo capturar una bandera, destruir un objetivo, o dominar áreas durante la mayor cantidad de tiempo. En la mayoría de los mapas, a los jugadores se les equipa con ciertos objetos que son otorgados cada vez que el personaje muere y reaparece. Estos *kits* suelen contener armas, herramientas, bloques para construir, comida, u otros objetos útiles para alcanzar el objetivo de ese mapa, los cuales se posicionan en el inventario del jugador en un orden por defecto, definido por el mapa.

Fig. 1: Kit simple y Kit complejo, y el orden de un usuario

El inventario del jugador en el juego tiene un total de 36 huecos (4x9), como se ve en la Fig. 1, siendo la fila inferior la única accesible (con las teclas 1 a 9 por defecto, o con la rueda del ratón) sin abrir el inventario, por lo que en estas posiciones se ponen los objetos que se usan con más frecuencia, mientras que el resto de objetos requieren abrir el inventario y moverlos con el ratón a la barra principal.

Solo es posible seleccionar un objeto en cada momento: para pelear, seleccionas la espada; para construir, los bloques; si caes o te empujan de un lugar alto, el cubo de agua puede salvarte la vida ya que caer en agua elimina el daño por caída. Cambiar de objetos de manera rápida y consistente es fundamental para progresar en el modo de juego, y el progreso adecuado en el juego dependerá de los objetos que tengas colocados en las posiciones 1 a 9 del inventario. Cada jugador tiene un estilo de juego, cada mapa tiene diferentes objetivos, y dependiendo del estilo y del mapa, cada jugador preferirá tener los objetos en unas posiciones u otras. Por ello, muchos jugadores reordenan los objetos que se les otorgan al principio de la partida, el problema es que ese orden se pierde cada vez que el jugador muere y reaparece. Esto, a petición de los usuarios, ha dado pie al desarrollo de este trabajo como solución al problema.

El objetivo general del proyecto es generar de forma automática *kits* ordenados según las preferencias de cada jugador para cada mapa. Para ello será necesario observar las modificaciones que hace a su inventario, aprender sus preferencias, y ser capaz de predecir cómo el jugador quiere su inventario en mapas futuros, con objetos distintos.

2 Estado del arte

Hay bastante literatura al respecto de perfilado de usuarios en videojuegos, desde como modificar el juego en tiempo real para optimizar la satisfacción de los usuarios [5] hasta predecir las posibilidades de ganar basandose en las composiciones de equipo [6]. Los diferentes tipos de perfilado del comportamiento de los usuarios ya han sido teorizados [7] y en este trabajo se discutirá un perfilado de jugadores de tipo individual y basado en datos.

3 PGM

PGM (PvP Game Manager)[2] es un *plug-in* para servidores de *minecraft*, que cambia como el juego se comporta de manera radical. En vez de ser un juego basado en la búsqueda de recursos y supervivencia, se basa en modalidades JcJ (jugador contra jugador) donde dos o más equipos compiten por ser el primero en alcanzar el objetivo, por ejemplo, conseguir una bandera, o dominar una zona durante el mayor tiempo posible. En la mayoría de mapas, los objetos principales son otorgados por PGM en forma de *kits*, que cada mapa define en un archivo de configuración. Hay cientos de mapas diferentes, algunos con kits similares, otros con kits distintos, y los objetivos así como la manera de jugar esos mapas, también varían enormemente.

La posición preferida de cada objeto es una decisión personal, muchos jugadores se adaptan al orden por defecto, pero siempre hay mapas donde ese orden no es el preferido, ya que uno tiene interiorizados ciertos atajos de teclado a ciertos objetos (por ejemplo, *1* para la espada, *5* para los bloques), y cambiarlo cada vez que tu personaje muere es muy tedioso. También es común que los jugadores con inventarios ordenados de manera poco convencional cambien las teclas más lejanas como *789*, ver Fig 2, a atajos de teclado más cercanos a *wasd* (las cuales se usan para mover el personaje) como *rfcv*. Por ejemplo, un jugador puede usar *f* para seleccionar la posición 8, y tener ahí el cubo de agua (como en kit simple de Fig. 1). Cuando necesita el cubo, presiona la tecla *f*, más cercana y accesible que *8*. Girar la rueda del ratón también cambia el objeto seleccionado, y funciona de manera circular (tras el *9* vuelve al *1*, y viceversa en sentido contrario), no se suele usar ya que es más lento que usar teclas específicas; sin embargo, sí es común acceder a la posición *9* seleccionando *1*, y girando la rueda para pasar directamente del 1 al 9.

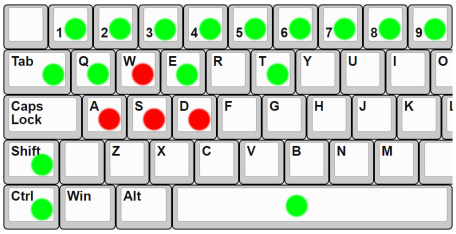

Fig. 2: Teclado qwerty, en verde las teclas asignadas a alguna función por defecto, en rojo *wasd*, usados para delante (w), izquierda (a), detrás (s) y derecha (d), muy común en muchos videojuegos 3D

3.1. Dificultades añadidas

A pesar de que la mayoría de los objetos suelen darse al reaparecer en forma de *kits*, a menudo a lo largo de la vida en la partida puedes obtener otros objetos, o gastar los objetos del *kit*. Por ejemplo, puedes llegar a una zona del mapa donde puedes conseguir un arco mejor, y posicionas el nuevo arco donde solía estar el anterior, y el anterior lo mueves a otra posición. El movimiento del arco anterior no puede considerarse una “preferencia” del usuario, ya que ha sido por una razón concreta (una mejora). Al reaparecer no querrás que ese cambio se mantenga. También es

posible obtener mas objetos al conseguir eliminaciones, o gastar los objetos del kit inicial, reemplazando las posiciones de objetos que has gastado con otros objetos, en esos casos también hay que tener en cuenta que esto puede ser motivado por otra razón, no una "preferencia" del usuario. Todo esto genera problemáticas a la hora de intentar inferir las preferencias del usuario, y añaden cierto ruido a las predicciones.

3.2. Tipos de kit

Existen diferentes tipos de kits dependiendo de los mapas, y muchas cosas pueden aplicar a unos mapas pero no a otros. A pesar de esto, suele haber algunos kits genéricos muy usados (aunque todos ellos, con alguna que otra variación), que son los siguientes:

Fig. 3: *Kit de conquest (conquista)*

Fig. 4: *Kit de CTW (capture the wool)*

Fig. 5: *Kit de DTC/M (destroy the core/monument)*

Los objetos que se ven en las Fig. 1 (primer kit), 3, 4 y 5 corresponden a las categorías: espadas (piedra, hierro y diamante), arcos (uno de ellos con encantamiento de infinidad, el cual no gasta flechas), comida especial (manzanas de oro), flechas, picos (hierro y diamante), hachas (hierro y diamante), pala (hierro), bloques (madera, cristal, ladrillos de piedra, y escaleras de mano), comida (zanahorias doradas y filetes), y finalmente, cubos (agua). Todos estos *kits* tienen muchas variaciones distintas, por ejemplo, la Fig. 1 corresponde a un mapa de tipo *conquest*, pero en el cual se puede construir, por lo que incluye herramientas, bloques, cubo de agua, y comida, que el kit en la Fig. 3 no necesita. Ningún objeto es exclusivo a un tipo de mapa, todos se pueden usar en cualquier mapa a discreción del autor, y esa es la razón por la que hay tanta diversidad entre los *kits*.

Por otro lado, el *kit* complejo de la Fig. 1 es específico a mapas con mecánicas del juego que involucran explosiones. Todos los objetos tienen utilidades concretas, pero son difíciles de explicar sin entrar en mecánicas muy específicas del modo de juego.

4 Objetivos

El objetivo general del proyecto es generar de forma automática *kits* ordenados según las preferencias de cada jugador para cada mapa. Para ello será necesario observar las modificaciones que hace a su inventario, aprender sus preferencias, y ser capaz de predecir cómo el jugador quiere su inventario en mapas futuros, donde los objetos serán distintos. Actualmente los usuarios no están incitados a editar su inventario porque lo pierden al morir y reaparecer, por lo que el primer paso (previo a la recolección) deberá ser convencer a los usuarios de que editar el inventario no es en vano, guardándoles los cambios para el mapa en concreto.

Los objetivos se separan en los siguientes pasos:

1. Crear un plugin que incite a los usuarios a editar sus kits, para poder recopilar y agregar datos reales en un *dataset* con el que analizar el problema (sección 6).

2. Estudiar los datos del *dataset* recopilado para sacar conclusiones sobre qué es relevante a la hora de predecir los kits (sección 7).

3. Diseñar e implementar diferentes modelos y arquitecturas para resolver el problema de la generación de kits automáticos (sección 8).

4. Probar y evaluar los diferentes modelos y resultados, rediseñando si fuera necesario (sección 9).

5. Implementar un producto final que interopere con PGM, capaz de reordenar los kits de los usuarios en producción, y analizar los resultados finales (sección 9.4).

6. Redacción de este informe, presentación, y demás defensa del trabajo

La solución final no deberá requerir un gran volumen de datos por jugador, ya que el espacio de almacenaje del servidor es limitado.

5 Planificación

La planificación del TFG ha sido la siguiente:

- Objetivo 1: Creación del dataset
  - Semana 1: Crear versión inicial del *plug-in* en java, crear un algoritmo de predicción de preferencias básico, y guardar las preferencias de usuario durante la partida en curso.
  - Semana 2: Añadir un sistema de categorización de objetos, definir la codificación de los objetos y definir los meta datos a guardar para cada categoría.
  - Semanas 3-4: Adaptar la librería *parquet-floor*[8] para migrar de java 11 a 8 (versión requerida por el servidor). Usar las librerías oficiales (en java 8) no es viable, ya que obligan a instalar un entorno de *hadoop*[9], y el *plug-in* debe tener mínimo impacto en el servidor.
  - Semana 6-7: Generalizar la implementación en java del algoritmo de predicción de preferencias básico, para poder aplicarlo a datos ya recolectados fuera del *plug-in*, en forma de programa independiente, que agregue los datos de participaciones individuales creando un *dataset* con datos más limpios y adaptados.
- Objetivo 2: Estudio de los datos

    - Semana 5: Leer los archivos parquet creados en java desde python, con las librerías pandas, y pyarrow o fastparquet. Re-implementar librerías implícitas de *bukkit* (el *framework* para *plug-ins* en *minecraft*) que no estarán presentes en python, como los tipos de materiales, encantamientos y otros meta datos.
    - Semana 7: Análisis inicial de los datos recopilados hasta el momento.
    - Semana 8: Replicar el sistema de categorización creado en java, para poder descodificar en python los meta-datos específicos de cada objeto. Hacer un análisis de los datos más exhaustivo, para poder determinar la forma real que tienen, y posibles categorizaciones necesarias para no crear modelos con *bias* hacia tipos de usuarios o tipos de kits concretos.

- Objetivos 3 y 4: Diseño, implementación y pruebas de modelos
    - Semana 9-14: Probar y entrenar diferentes modelos, empezando por mantener los kits por defecto como referencia, y continuar con otras alternativas descritas en la sección 8. Estos modelos se entrenarán con una segunda versión de los datos, y el análisis de la semana 8 se revalidará con los datos actualizados, y se expandirá el más gráficos y métricas necesarias para entender porqué los modelos generan los resultados que generan. Por último se generarán resultados provisionales y visualizaciones para entender lo bien que están funcionando.

- Objetivo 5: Diseño, implementación y pruebas de modelos
    - Semana 15: Implementar el modelo con mejor rendimiento y menor cantidad de datos por usuario requerido en java como parte del *plug-in*, definiendo el modelo de datos a guardar por usuario y su arquitectura, desplegándolo en producción, y recabando *feedback* de los usuarios.

- Objetivo 6: Creación del articulo y la presentación
    - Semanas 16-17: Redacción del informe final para presentar a revisión.
    - Semanas 18-20: Preparación de la presentación, defensa final del trabajo, y pulir el informe final. No se considera la creación de póster.

La cronología ha sido modificada a lo largo del proyecto, en concreto se han añadido los pasos de procesado 6.2 y análisis de datos 7, para simplificar las entradas de datos a los futuros modelos y entender mejor los datos con los que estamos tratando, además de filtrarlos para eliminar futuro ruido en los resultados y reducir el *bias*.

También ha requerido más tiempo del planeado poder leer los archivos *parquet* desde *python* por problemas de compatibilidad, y ha requerido hacer una transformación extra en *java* para convertir de *parquet* 2.0 a 1.0, el cual tiene mayor compatibilidad [10].

6 Generación del dataset

6.1. Recolección de los datos

Para poder averiguar el orden en el que un jugador quiere un kit, debemos observar como lo reordena cuando se le otorga. Para tener esa información necesitamos un *dataset* sobre el cual poder probar diferentes modelos de aprendizaje automático.

Intentar recompilar tales datos es complejo, porque los usuarios no van a darnos la información de manera voluntaria, sería demasiado tedioso pedir a gran escala que los usuarios reordenasen varios kits distintos para poder hacer tal análisis. Actualmente en el servidor, el kit se otorga cada vez en el orden por defecto, lo que implica que para jugar con un kit modificado, se deberán mover los objetos cada vez que muera y reaparezca un jugador. Esto no es algo que muchos usuarios estén dispuestos a hacer, y lo que ocurrirá es que jugarán con los kits por defecto la gran mayoría de las veces solo para no tener que modificar los objetos. Para poder lograr recopilar los datos, necesitamos conseguir cambiar la forma actual de jugar, y de alguna manera "convencer" a los jugadores de que deberían hacerlo.

Con el objetivo de acostumbrar a los usuarios a reordenar el inventario, se ha desarrollado un nuevo *plug-in* que se integra con *PGM* y cuya primera versión (desplegada el 15 de febrero) deduce el orden preferido con algoritmos simples (solo para la partida actual) y aplica ese orden al reaparecer sin almacenar datos.

Se ha tenido en cuenta el *feedback* de los jugadores, que mencionaban problemas descritos en la sección 3.1, para desarrollar la segunda versión (desplegada el 18 de febrero) que mejora el algoritmo introduciendo heurísticas para fomentar la complicidad de los jugadores y que estos confíen y se acostumbren a usar el sistema, sin generar rechazo hacia él, consiguiendo fomentar que los usuarios ordenen su inventario.

El algoritmo actúa en el momento que el usuario cierra su inventario, el orden actual intenta aplicarse al *kit* original, haciendo una búsqueda de los objetos y cambiando las posiciones para seguir las preferencias. Solo se consideran preferencias (y por lo tanto, se guardan) movimientos que implican objetos presentes en el kit original, ya que reemplazar un objeto por uno no presente en el kit no implica que se quiera guardar esa preferencia. Por ejemplo, si se reemplaza la espada de madera del kit por una de hierro que se ha conseguido en el mapa, no implica que se quiera la espada de madera en una posición arbitraria del inventario, solo significa que la ha reemplazado por una alternativa en la misma posición; donde acabe la espada de madera es irrelevante.

La tercera versión (desplegada el 2 de marzo) añade la recopilación y el almacenaje de datos de usuarios para cada *muestra*. Una muestra se da cuando, o el usuario recibe un kit por el programa (en este caso *closed* sería false), o el usuario abre su inventario, lo modifica (o no) y lo cierra (en este caso *closed* sería true). Se ha de tener en cuenta que para modificar el inventario es necesario abrirlo y cerrarlo.

Cada muestra tiene los siguientes atributos:

- *timestamp*: el instante de tiempo.
- *closed*: *true* para inventario cerrado, *false* para kit otor-

gado.

- *slot_0 - slot_35*: el objeto en cada hueco de inventario, codificado en un *int32* con 3 partes de 16, 8 y 8 bits: material[0..15], cantidad[16..23], y meta datos[24..31]

El resultado de este proceso genera una carpeta por cada usuario, y un archivo por cada partida. Usar un formato que ocupe poco espacio es esencial para poder recolectar un gran volumen de datos sin llegar a los límites de disco, por lo que se ha decidido usar el formato *parquet* [11]. Este formato está asentado en la industria, con compatibilidad en varios lenguajes de programación, ocupa poco espacio en disco, y es rápido de leer y escribir. En 54 días (del 2/3/2022 a 25/4/2022), se crearon 327.997 participaciones, ocupando 2,6GB. Esta recolección de datos es viable para el estudio del problema, pero no será viable a largo plazo una vez esté implementada la solución final.

6.2. Reducción de dimensionalidad

La información guardada era muy extensa y detallada, y genera un problema a la hora de intentar analizarla. El tamaño en disco no es un problema, pero es muy complejo intentar sacar conclusiones sobre unos datos tan extensos y específicos, para poder simplificar el problema necesitamos generalizar y extraer datos más concretos. Por esa razón, se ha aplicado una capa de procesado de datos en la cual se infiere la preferencia del usuario a partir de los datos, reduciendo así la dimensionalidad de estos.

Con esta transformación pasamos de tener *#usuarios * #partidas * #cambios de inventario* a simplemente *#usuarios * #partidas*, donde cada partida tiene el kit original y el kit final o "preferencia" del usuario.

Por las razones descritas en la sección 3.1, no podemos simplemente mirar la última fila del archivo y tomarla como preferencia, ya que es posible que ese inventario no contenga muchos de los objetos del kit original, o que contenga otros distintos, y necesitamos que el kit original y el kit final tengan exactamente los mismos objetos, solo modificando el orden, ya que esto es lo único que podemos cambiar (no podemos generar objetos nuevos, o eliminar objetos una vez esté en producción).

La preferencia de usuario es inferida a partir de los cambios individuales que el usuario hace a lo largo de la partida. Para ello se toma como correcta la última posición del objeto en la partida, siempre y cuando este no haya sido reemplazado por un objeto que no existía en el kit original. El algoritmo que crea *posiciones finales* que serán posteriormente usadas como *ground truth* es el mismo algoritmo simple usado para reordenar objetos dentro de una partida por el *plug-in*, mencionado en apartado 6.1.

Los datos *raw* pasan a ser un solo archivo por usuario. Cada archivo contiene una fila por cada partida, la cual incluye el kit original y la preferencia del usuario. Los archivos finales incluyen las siguientes 74 columnas:

- *timeframe_seconds*: el tiempo (en segundos) desde la primera hasta la última *muestra*, definida en la sección 6.1. Puede usarse para filtrar casos donde el usuario pasó poco tiempo en la partida y/o dar mas peso a partidas largas, pero finalmente no ha sido utilizado.
- *edited_inventory*: *true* si el usuario llegó a abrir y cerrar el inventario tras recibir el kit, si se ha movido algún objeto es irrelevante, *false* en caso contrario. Podría usarse para descartar casos, pero finalmente no ha sido utilizado.
- *kit_0 - kit_35*: el objeto en cada hueco de inventario tras recibir el kit. Codificado igual que *slot_**, ver 6.1.
- *sorted_0 - sorted_35*: el objeto en cada hueco de inventario tras aplicar los cambios de preferencia del usuario. Codificado igual que *kit_**.

Tipo	Casos	Descripción
Corruptos	72	Archivos sin escribir cuando se aplicó el procesado.
Sin kit	13	Sin otorgar *kit*, pero el jugador abrió su inventario.
Kits distintos	3.533	Otorgados distintos *kits* en una partida, no hay una preferencia única y fiable.
Objeto único	7.924	El *kit* tiene un único objeto.
Muestra única	32.464	El archivo tiene solo una *muestra*.

TABLA 1: CRITERIOS Y CASOS DE DESCARTE DE PARTIDAS AL PROCESAR LOS DATOS INICIALES.

Durante el procesado se ignoran partidas por falta de datos, las causas concretas y casos encontrados están descritos en la Tabla 1, mayoritariamente se ignoran archivos con una única muestra, en los cuales se ha otorgado un kit pero nunca llegaron a modificarlo, ni reaparecieron para que se les otorgase una segunda vez. De esta manera, las 13.161 carpetas con 327.997 archivos, pasan a ser 13.161 archivos con 283.999 preferencias de *kit*, y 44.006 participaciones son descartadas.

7 ANÁLISIS DE DATOS Y ESTADÍSTICAS

Ha sido necesario llevar a cabo un análisis de los datos recolectados para saber, entre otras cosas, de cuántos usuarios disponemos, cuántos *kits* distintos hay en el *dataset*, o con qué frecuencia lo usuarios editan su inventario.

Como se puede observar en la Fig. 6 disponemos de pocos usuarios con muchas partidas, y muchos usuarios con pocas partidas. Existe una pequeña élite de unos 20 jugadores con más de 1000 partidas, y una mayoría de más de 10.000 con menos de 10 partidas. Teniendo en cuenta las conclusiones inferidas del gráfico y con el objetivo de reducir el *bias* del dataset y por motivos de rendimiento, se ha decidido prescindir de todos los usuarios con menos de 10 partidas al entrenar modelos.

Una vez aplicado el filtro de usuarios, el total de preferencias del *dataset* pasa de 283.999 a 259.642 y el total de jugadores baja de 13.161 a 3.050. Esto simplifica el procesado de datos al bajar la cantidad de jugadores a menos de una cuarta parte, y reduce el ruido que los usuarios con muy pocas preferencias introducen, los cuales además representarían una mayoría si se agregan datos por jugador sin dar más peso a los jugadores con mas casos.

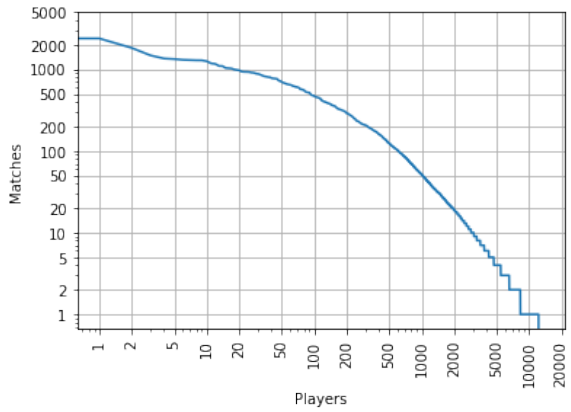

Fig. 6: Número de participaciones por usuario. Ambos ejes en escala logarítmica.

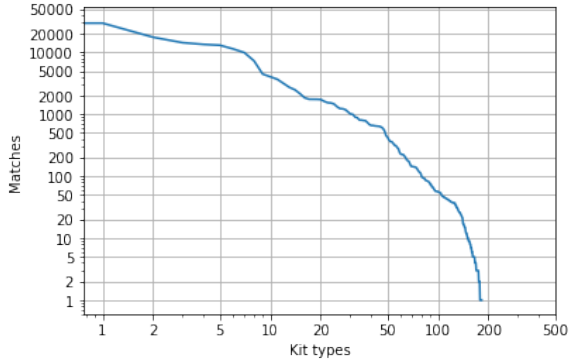

Fig. 7: Número de casos por tipo de kit. Ambos ejes en escala logarítmica.

Por otro lado podemos observar distintos tipos de kits. El *tipo de kit* se define por el conjunto de categorías de cada objeto del kit, ignorando el orden y las repeticiones. Hay *kits* estándar, mientras que otros *kits* tienen muchas variaciones de materiales y/o herramientas, resultando en diversos *tipos de kit*, ver 3.2. Analizando los datos por *tipo de kit*, ver Fig. 7, se aprecia que hay algunos pocos *tipos de kit* genéricos, que son muy comunes en muchos mapas, y que tienen muchos usos; por otro lado, hay muchos *tipos de kits* minoritarios, utilizados en pocas participaciones, que pertenecen a mapas específicos con un carácter más único, que están en pruebas, o que tal vez tienen un error en el kit (e.g: falta una herramienta) que se subsana al poco tiempo, y que por lo tanto, no cosechan una gran cantidad de usos. Por una razón similar a la expuesta anteriormente para los jugadores, también se van a eliminar los kits cuyo *tipo* no cuente con al menos 50 usos, creando así varios datasets filtrados, más limpios.

Para analizar cuán a menudo los usuarios editan el kit que se les otorga y en que medida, para cada usuario, se calcula el porcentaje de *kits* donde algún objeto se ha movido y el porcentaje de objetos movidos. Estos valores se reflejan respectivamente en los ejes X e Y de la Fig. 8.

Los histogramas resultantes muestran que lo más común es mover pocos objetos (entre 0 % y 10 %); sin embargo, la mayoría de usuarios sí modifica alguno de los *kits* por defecto, solo el 15.2 % (464 de 3050) de los jugadores editaron menos del 10 % de los *kits*. Aún así la mayoría de usuarios modifica un porcentaje bajo de los *kits*.

Los datos sugieren que posiblemente hay una correlación entre el *kit* otorgado y si el usuario lo edita o no. Esto

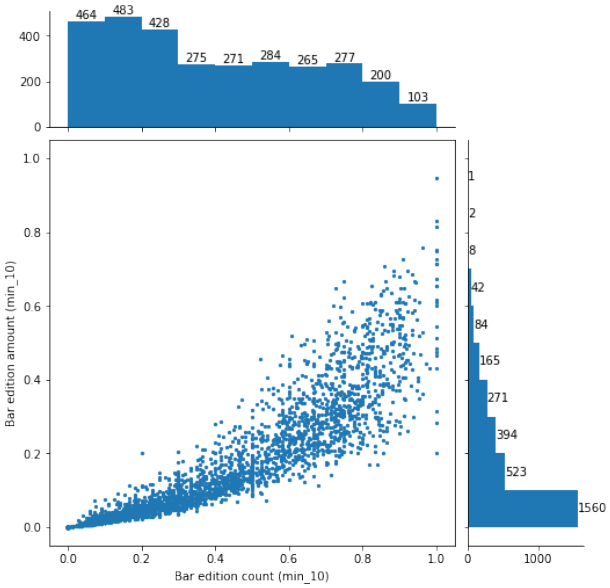

Fig. 8: Cada punto representa un jugador, cuantos objetos se han movido en el eje vertical, y cuantos *kits* se han editado en el horizontal. Solo incluye cambios a la barra principal, y solo usuarios con al menos 10 partidas, ver 9.1

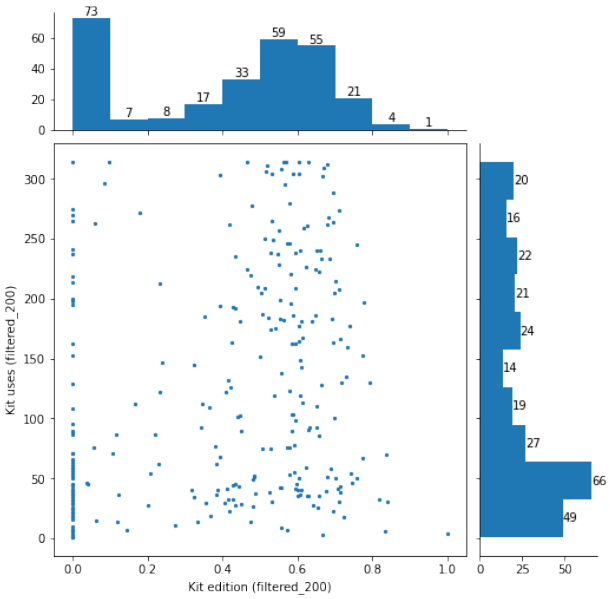

Fig. 9: Cada punto representa un kit. En el eje vertical, cuantos jugadores han usado este kit; en el horizontal, el porcentaje de las veces que se ha otorgado y ha sido modificado. Solo incluye cambios a la barra principal. Solo contiene *kits* con al menos 50 del mismo tipo. Solo usuarios con al menos 200 partidas, ver 9.1

explicaría la distribución semi-uniforme del porcentaje de *kits* editados, considerando que los mapas jugados por cada usuario son aleatorios y distribuido uniformemente. La Fig. 9 también apoya esta hipótesis, mostrando que algunos kits son editados más que otros, formando aparentemente una distribución normal.

8 Modelos y arquitecturas

Hay que tener en cuenta que la versión para producción deberá requerir la mínima cantidad de datos persistidos por usuario, para poder escalar sin costes significativos. Como

referencia sabemos que el volumen de jugadores en la semana del 20 al 27 de febrero de 2022 fue de 4.655 jugadores únicos, que crearon 23.445 sesiones, con una media de 19m 32s por sesión (≈317 días de tiempo de juego). Por lo que modelos donde para cada usuario hay que guardar muchísimos datos no serán viables para producción.

Los modelos tomarán como input el kit a otorgar, y deberán dar como salida el inventario preferido. Los modelos serán entrenados para cada usuario individualmente. Se considera que sería posible crear modelos genéricos para grupos de usuarios, o por tipos de mapas, pero se deja como trabajo futuro.

Se han implementado un total de 3 modelos, basados en la idea de los modelos de naive bayes. Todos ellos funcionan sobre un jugador específico. Los modelos responden la pregunta de "para un usuario, cúal es la probabilidad de que quiera un objeto en una posición" y un componente genérico para los 3 genera el kit final de manera iterativa, buscando que objeto y posición tiene probabilidad mas alta, posicionándolo, eliminando esa posición de todos los demás objetos, y pasando al siguiente. Todos los modelos trabajan sobre categorías, no objetos concretos (ver 3.2). Los 3 modelos, llamados v1, v2 y v3 respectivamente, difieren en su funcionamiento interno, que es el siguiente:

- v1: Para cada categoría guarda una matriz de 1x9 posiciones. Cada posición describe cuantas veces la categoría ha acabado en esa posición, ya sea por que es la posición inicial, o porque el usuario la ha movido. Para calcular la probabilidad, se selecciona la fila y se divide por el total de casos.

- v2: Para cada categoría guarda una matriz 2x9. La primera fila es usada para los objetos no movidos (posición en el kit inicial y kit final es la misma), y la segunda para los movidos. Esto implica que la primera fila corresponde a "el jugador no ha movido el objeto" mientras que la segunda fila corresponde a "el jugador ha movido el objeto a esta posición". Para calcular la probabilidad, se selecciona la casilla de la primera fila o la segunda fila dependiendo de si el kit original tiene el objeto en esa posición o no, y luego se divide por el total.

- v3: Para cada categoría guarda una matriz 10x9, la Fig 13 es un ejemplo de esto. Cada fila corresponde a la posición donde se ha dado el objeto, la última fila representa las posiciones 10 a 36 (objetos dados fuera de la barra principal). Cada columna las veces que el objeto ha acabado en tal posición. Por lo tanto, cada casilla (*x*,*y*), las veces que un objeto dado en posición *min(y, 10)* (limitado a 10 en caso de ser superior) ha acabado en posición *x*.

  Para calcular la probabilidad, se selecciona la fila correspondiente a la posición inicial, y se divide por el total de la fila. Dado PF = Posición final, PI = Posición inicial, C = Categoría, y U = Usuario, podemos describir el comportamiento bayesiano del modelo de la siguiente manera:

$$P(PF|PI, C, U) = \frac{ProbConj(PI, PF, C, U)}{P(PI, C, U)}$$

  El modelo también tiene en cuenta si una posición final puede venir de varias posiciones iniciales, en cuyo caso asigna una facción correspondiente. Ejemplo: Kit original tiene bloques en posiciones 6 y 7, el kit preferido tiene bloques solo en la posición 2, en la matriz las posiciones [6,2] y [7,2] suman +0.5, como "medio objeto" se ha movido de 6 a 2, y otro medio de 7 a 2. En caso que hubiese 2 posiciones finales, al ambas sumar 0.5 acabaría siendo equivalente a que se tratasen los movimientos de manera independiente (sumando uno cada uno), pero esta manera de tratarlo genera una distribución mas justa en caso de que no cuadren los objetos finales con los iniciales.

9 EXPERIMENTOS

9.1. Dataset

El dataset consiste de un archivo por cada usuario, con una fila por mapa, teniendo 74 columnas que contienen algunos datos generales de la partida, además de los 36 huecos del inventario tras otorgar el primer kit, y los 36 huecos del inventario con el kit ordenado con las preferencias del usuario, más detallado en la sección 6.2.

Para facilitar los experimentos, se generan varios sub datasets. Las razones se detallan en la sección 7:

- *all*: Contiene todos los usuarios y preferencias de kit (el dataset tras la reducción de dimensionalidad, completo)

- *min_n*: Contiene los usuarios con al menos n partidas, con todas sus preferencias de *kit*

- *filtered_n*: Contiene los usuarios con al menos n partidas, eliminado previamente los *kits* poco comunes (menos de 50 kits del mismo tipo en todo el dataset).

Los datasets con n, tienen varios tamaños, 10, 35, 50, 100 y 200, para poder probar los modelos de manera más rápida en los datasets con menos usuarios (*filtered_200*) y luego ejecutando con las versiones con más datos.

No se considera hacer splits de train, test y validación como en la mayoría de proyectos de ML, ya que se considera que los datos llegarán en tiempo real, es decir, una vez el sistema esté desplegado, tendrá que predecir y aprender simultáneamente, y ese es el comportamiento que vamos a simular para evaluar los modelos.

9.2. Metricas

La métrica principal para evaluar el rendimiento de los modelos contabiliza la cantidad de objetos que han sido posicionados correctamente, comparando la predicción del modelo y el *ground truth*. Se entiende como error el porcentaje de objetos que no están en la misma posición que el *ground truth* para cada kit.

No se considera usar métricas de distancia de edición [12], ya que los objetos deben estar exactamente en la posición que el usuario desea, no en una cercana. Lo que parece lineal (del 1 al 9) no necesariamente lo es, ya que los jugadores con los inventarios más peculiares suelen ser los que cambian los atajos de teclado (comúnmente el 7, 8 y 9), como se explica en la sección 3. En un caso como el del kit

simple de la Fig. 1, un usuario tiene el cubo de agua en posición 8, y configura sus atajos para que la tecla *f* seleccione el 8. Poner el cubo en cualquier otra posición se tiene que considerar un fallo, independientemente de lo cerca del 8 que esté.

9.3. Resultados teoricos

Los tres modelos han sido entrenados con los datos del dataset *filtered_200*, que corresponde a usuarios con al menos 200 partidas y excluyendo algunos *kits* muy poco comunes que pueden añadir ruido, ver 9.1.

El error por defecto, consiste en no aplicar ninguna modificación al *kit* por defecto y observar qué error ocurre, es decir, cuantos objetos son movidos por el jugador. Se usa como *baseline* para comparar con otros modelos, cuantos objetos tiene que mover por defecto en comparación a cuántos objetos debe modificar si los otorgamos ordenados por un modelo.

Como se puede observar en la Fig. 10, el modelo v1 es demasiado simple para modelar las preferencias de los usuarios, por lo que en la mayoría de casos genera peores resultados que los kits por defecto, ya que una gran parte de los usuarios usa el orden por defecto, o similar. El modelo v2 es el primero que consigue una rebaja de error medio respecto a los kits por defecto, pero causa a algunos usuarios que utilizan kits por defecto, o similares a por defecto, un error mayor. El modelo v3 es el que por fin consigue resultados significativamente mejores que por-defecto en la mayoría de casos, y muy similares a por-defecto en el resto. Sin embargo hay un dato que la Fig. 10 no recoge, y es cuanto ha mejorado o empeorado cada caso, ya que cada una de las series ha sido ordenada, por lo que un mismo punto en el eje x de diferentes series no corresponden al mismo jugador.

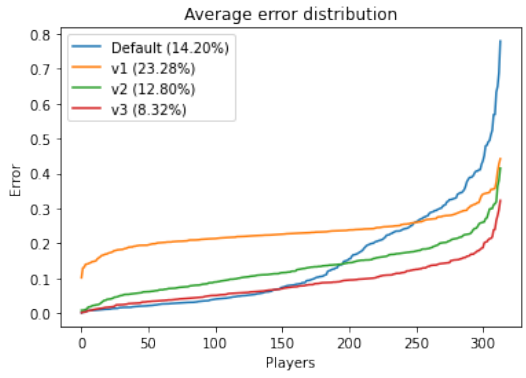

Fig. 10: Error medio para un jugador concreto en el eje vertical, cada serie está ordenada de menor a mayor

Para recoger este dato, hemos generado la Fig. 11, que representa para para cada usuario, el calculo de la diferencia entre por-defecto y cada modelo. Aquí podemos observar claramente cuanta mejora genera para cada usuario cada uno de los modelos. En el caso del modelo v3, aproximadamente la mitad de los usuarios cosechan mejoras, y son muy pocos los usuarios que consiguen unos errores significativamente peores que por defecto (al rededor del -5 % en los peores casos).

Por último hay otro dato subyacente que las Fig. 10 y 11 no muestran, y es qué perfil de usuario es el que cosecha mayores ganancias o perdidas. Para ello tenemos la Fig 12 la cual muestra las transferencias entre grupos para el modelo v3. Aquí se puede apreciar que todos los usuarios que cosechan peores resultados corresponden a usuarios que mantienen el kit por defecto, y la gran mayoría de usuarios con kits alejados del por-defecto cosechan mejores resultados que por defecto.

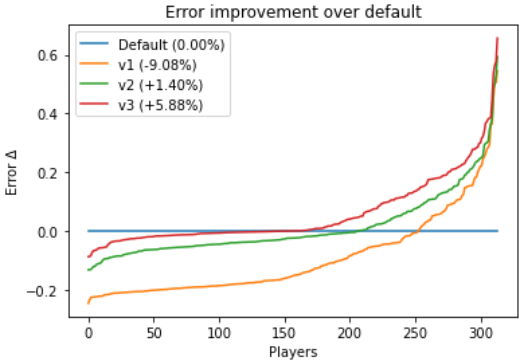

Fig. 11: Series ordenadas ascendentemente que representan la diferencia entre el error por defecto y error en el modelo concreto, para cada usuario. Los puntos por encima de 0 suponen una mejora respecto al kit otorgado por defecto.

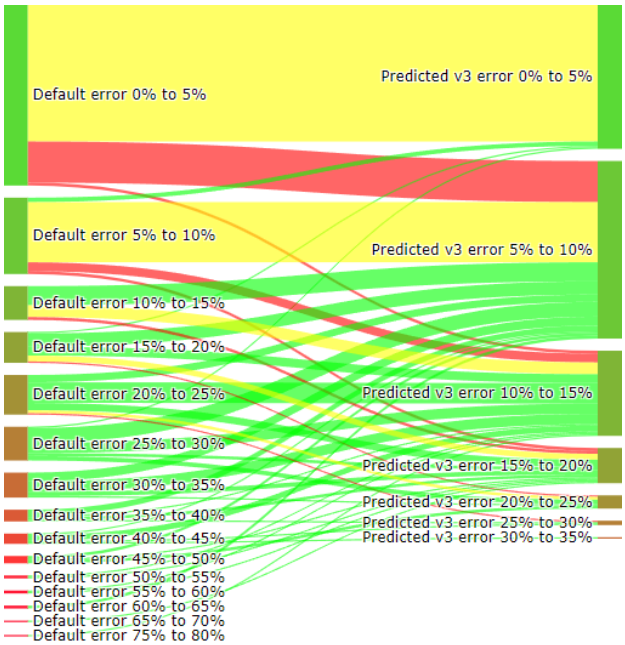

Fig. 12: Diagrama de flujo del error de los jugadores, a la izquierda dejando el kit por defecto y a la derecha aplicando las predicciones del modelo v3. Los usuarios han sido agrupados en rangos de 5 % y el tamaño vertical es proporcional al número de usuarios en el grupo.

Todos estos datos de errores deben ser tomados en contexto, ya que cualquier cambio que el usuario no haga se considera error, a pesar de que esto podría ser lo que el usuario en realidad quería, solo que no lo ha hecho (por ejemplo, por pereza). Un ejemplo sería un usuario que cambia un objeto el 80 % de las veces, el modelo predice siempre esa nueva posición, ya que es la más probable; sin embargo, esto genera un error del 20 % en los resultados, a pesar de que el usuario en realidad lo prefiere así, solo que en algunos casos no se ha molestado en cambiarlo. El análisis de casos

individuales confirma que ocurre, pero determinar en que medida esto empeora el error medio es difícil de predecir.

Como último resultado interesante, podemos observar que si indagamos en los datos internos del modelo v3, podemos extraer información relevante para la sección 7 en las Fig. 13 y 14. En estas tablas lo que se puede leer es como la diagonal (dejar la posición por defecto) es lo más común. Esto sobre todo ocurre para la primera columna, donde suele estar posicionada la espada, o arma principal, y prácticamente nunca se mueve a otra posición (98.8 %). Las posiciones 5 en adelante mantienen la posición por defecto menos a menudo (75-80 %), ya que a partir de aquí es donde hay menos consenso, tanto en los jugadores como en los kits por defecto de los mapas, ya que el kit por defecto que se aplica en el mapa depende del autor, ver 3.2.

	Final #1	Final #2	Final #3	Final #4	Final #5	Final #6	Final #7	Final #8	Final #9	Sum
In #1	141.709	857	432	98	60	71	55	86	49	143.417
In #2	838	131.644	6009	929	915	965	400	268	250	142.218
In #3	364	3.322	120.033	2.916	3.992	1.590	1.604	548	453	134.822
In #4	160	622	1664	95.411	3804	3233	1709	1061	1414	109.078
In #5	50	1.909	2.012	2.143	72.680	1.211	4.273	5.070	7.070	96.418
In #6	34	1.000	1.162	2.049	946	77.575	2.607	3.764	5.339	94.476
In #7	27	309	907	1.015	2.583	1.200	52.295	3.711	4.318	66.365
In #8	22	279	493	981	2.203	1.960	3.025	34.083	2.641	45.687
In #9	29	711	837	2.291	7.682	4.882	5.842	8.382	93.208	123.864
#10-36	200	499	1.024	913	3.049	1.156	3.604	3.530	2.723	16.698
Sum	143.433	141.152	134.573	108.746	97.914	93.843	75.414	60.503	117.465	973.043

Fig. 13: Cantidad de veces que cualquier objeto del kit por defecto, en el dataset *filtered_200*, se ha dejado en posición final *Final #{1-9}* dada una posición inicial *In #{1-9}*.

	Final #1	Final #2	Final #3	Final #4	Final #5	Final #6	Final #7	Final #8	Final #9
In #1	98,81%	0,60%	0,30%	0,07%	0,04%	0,05%	0,04%	0,06%	0,03%
In #2	0,59%	92,56%	4,23%	0,65%	0,64%	0,68%	0,28%	0,19%	0,18%
In #3	0,27%	2,46%	89,03%	2,16%	2,96%	1,18%	1,19%	0,41%	0,34%
In #4	0,15%	0,57%	1,53%	87,47%	3,49%	2,96%	1,57%	0,97%	1,30%
In #5	0,05%	1,98%	2,09%	2,22%	75,38%	1,26%	4,43%	5,26%	7,33%
In #6	0,04%	1,06%	1,23%	2,17%	1,00%	82,11%	2,76%	3,98%	5,65%
In #7	0,04%	0,47%	1,37%	1,53%	3,89%	1,81%	78,80%	5,59%	6,51%
In #8	0,05%	0,61%	1,08%	2,15%	4,82%	4,29%	6,62%	74,60%	5,78%
In #9	0,02%	0,57%	0,68%	1,85%	6,20%	3,94%	4,72%	6,77%	75,25%
In #10	1,20%	2,99%	6,13%	5,47%	18,26%	6,92%	21,58%	21,14%	16,31%

Fig. 14: Equivalente porcentual a la Fig. 13, cada casilla se ha dividido por el total de la fila.

9.4. Resultados en producción

Una vez hemos validado los resultados en el dataset, lo último que queda es programar tal modelo en el *plugin* y desplegarlo en producción, guardando en un dataset distinto los datos posteriores al despliegue. Computando el error medio en ese dataset, podemos ver el rendimiento del sistema en el mundo real, y podemos ver resultados positivos.

Estos resultados corresponden a la intersección de los usuarios presentes en el dataset *filtered_200* utilizado en la sección 9.3, y los del dataset *filtered_35* de los datos de 3 semanas tras el despliegue, un total de 176 jugadores.

Como se puede observar en las Fig. 15 y 16, el error medio ha caído casi un 2 % más de lo esperado por el modelo teórico (convirtiendo una mejora teórica de x1,73 en una mejora del x2,26). Esto se debe principalmente a que, como comentábamos en la sección 9.3, un usuario puede no mover un objeto a una posición que le és mas cómoda por pereza, y en este caso si nosotros hemos predicho que el objeto iría en esa posición, no dará un error. Pero en el mundo

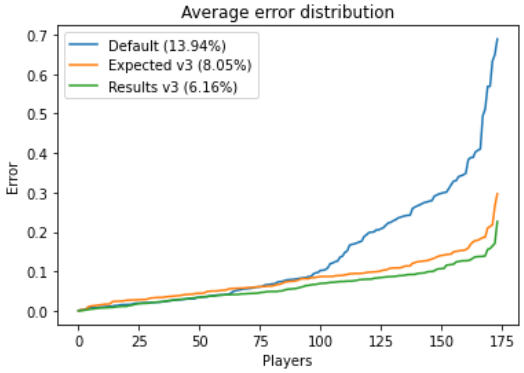

Fig. 15: Error medio para un jugador concreto en el eje vertical, cada serie está ordenada de menor a mayor. *Expected*: error teórico; *Results* error real en producción.

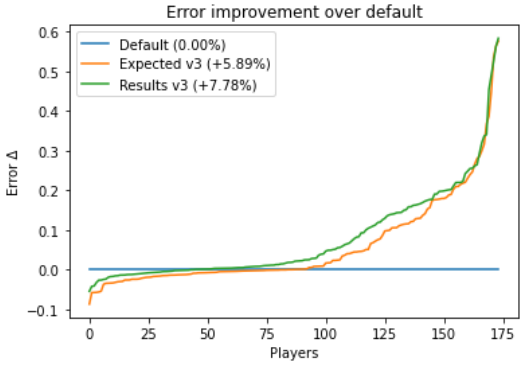

Fig. 16: Series ordenadas ascendentemente que representan la diferencia entre el error por defecto y error en el modelo concreto, para cada usuario. Los puntos por encima de 0 suponen una mejora respecto al kit otorgado por defecto. *Expected*: error teórico; *Results* error real en producción.

real vemos que si dejamos el objeto en la posición prevista, sería la correcta. Por esta razón el porcentaje de error en producción es un 2 % menor respecto al caso teórico, validando que era la posición real en la que quería el objeto.

Por último si nos fijamos en la distribución de usuarios que mejoran o empeoran los kits al cambiar de kits por defecto al modelo desarrollado, se ve que la inmensa mayoría si consigue una mejora, como se aprecia en la Fig. 17.

10 Conclusiones

Se ha desarrollado un predictor de kits, que dado un usuario y un kit genera una organización lo más amigable posible para el usuario, teniendo en cuenta lo que ha hecho en el pasado con sus kits. Se ha logrado una mejora del error por defecto (entendido como, cuantos objetos de los kits son movidos por los usuarios) de un x2.26. Para lograr tales objetivos se ha tenido que crear y analizar un dataset, diseñar, crear, analizar y mejorar distintos modelos de predicción, y acabar con un *plugin* en producción funcional que consigue aplicar el modelo y guardarlo para cada usuario.

Este TFG me ha servido para aprender a tratar con grandes cantidades de datos reales. Para hacerlo he visto que es necesario generar visualizaciones diversas, gráficos, y tablas para agregar la información, y lo que es aún mas importante, decidir como agregar los datos, porque hay muchas maneras de agregar y presentar los mismos datos, pero

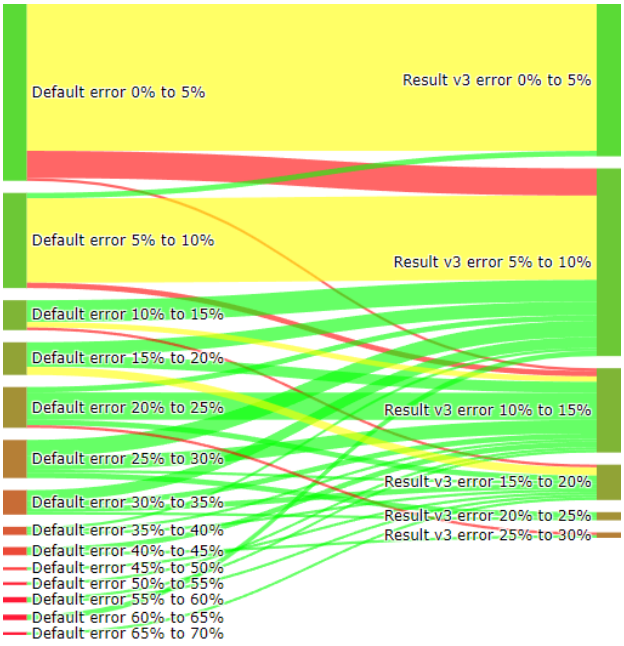

Fig. 17: Diagrama de flujo del error de los jugadores, a la izquierda dejando el kit por defecto y a la derecha aplicando el modelo v3 en producción. Los usuarios han sido agrupados en rangos de 5 % y el tamaño vertical es proporcional al número de usuarios en el grupo.

que no todas nos sirven para resaltar lo que es representativo de esos datos y extraer conclusiones.

11 Trabajo futuro

Hay varias vías para poder seguir mejorando este modelo, y la primera de ellas reside en una limitación directa del modelo v3, que es la falta de poder indicar que un objeto no lo quieres en una posición 1-9. En estos momentos el modelo no tiene una manera de guardar tal preferencia, por lo que en algunos casos, un objeto puede considerarse con un 100 % de probabilidad de querer ponerse en una posición, a pesar de que en la mayoría de los casos queda guardado dentro del inventario, porque todos esos casos no están contabilizados. Sería muy interesante ver un modelo v4 que tratase esa limitación.

Por otro lado, los datos que se han acabado aplicando a los modelos son limitados, solo se han utilizado las categorías (ignorando el tipo de ítem, la cantidad, y muchos otros meta-datos), y el único valor usado para predecir la posición final, era la posición inicial. Todo esto encaja dentro de un modelo de naive bayes, asume la independencia de variables que claramente no son independientes. Futuros modelos podrían tener en cuenta todo el resto de objetos en el kit para predecir donde será puesto el objeto.

Otro ejemplo donde el modelo actual está limitado, es el concepto de colocar un cierto objeto (o categoría) en "la primera posición libre" tras haber colocado antes las herramientas. Como el modelo para poner un objeto no tiene información sobre el resto de objetos, no puede hacer una predicción de este estilo de manera correcta. Lo que acaba ocurriendo con el modelo v3, es que esta categoría obtiene porcentajes bajos (repartidos entre varias posiciones), por lo que las herramientas (mas estáticas, y con porcentajes más altos) son colocadas primero, y luego de manera algo arbitraria, el primer hueco suele tener más probabilidad que el siguiente, por lo que el objeto se coloca correctamente, o en otras ocasiones no tiene tal probabilidad, y se pone dejando un o dos huecos vacíos.

Referencias

[1] *Minecraft official site*, feb. de 2022. dirección: https://www.minecraft.net/ (visitado 12-06-2022).

[2] PGMDev, *PGMDev/PGM: The original PVP game manager for Minecraft*. dirección: https://github.com/PGMDev/PGM (visitado 12-06-2022).

[3] C. A. Gomez-Uribe y N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, n.º 4, págs. 1-19, 2015.

[4] M. Cipollone, C. C. Schifter y R. A. Moffat, "Minecraft as a creative tool: A case study," *International Journal of Game-Based Learning (IJGBL)*, vol. 4, n.º 2, págs. 1-14, 2014.

[5] G. Yannakakis y J. Hallam, "Real-Time Game Adaptation for Optimizing Player Satisfaction," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 1, págs. 121 -133, jul. de 2009. DOI: 10.1109/TCIAIG.2009.2024533.

[6] H. Y. Ong, S. Deolalikar y M. Peng, "Player Behavior and Optimal Team Composition for Online Multiplayer Games," *CoRR*, vol. abs/1503.02230, 2015. arXiv: 1503.02230. dirección: http://arxiv.org/abs/1503.02230.

[7] Andersdrachen, *Behavioral profiling in games: An overview*, jun. de 2019. dirección: https://andersdrachen.com/2017/11/02/behavioral-profiling-in-games (visitado 12-06-2022).

[8] strategicblue, *strategicblue/parquet-floor: A lightweight Java library that facilitates reading and writing Apache Parquet files without Hadoop dependencies*. dirección: https://github.com/strategicblue/parquet-floor (visitado 12-06-2022).

[9] *Make it easy to read and write parquet files in java without depending on Hadoop*. dirección: https://issues.apache.org/jira/browse/PARQUET-1126 (visitado 12-06-2022).

[10] Dask, *Incorrect values being read from parquet file · issue 766 · Dask/Fastparquet*. dirección: https://github.com/dask/fastparquet/issues/766 (visitado 12-06-2022).

[11] D. Vohra, "Apache parquet," en *Practical Hadoop Ecosystem*, Springer, 2016, págs. 325-335.

[12] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*,