Implementación de una técnica de aprendizaje máquina sin el uso de un framework.

Imanol Muñiz Ramirez A01701713

Contents

Introducción

Contexto

Teamfight Tactics es un juego online de 8 jugadores que consiste en construir el equipo más fuerte para derrotar al de los demás. Cada ronda se simula el enfrentamiento y obtienes monedas de acuerdo con los resultados. Entre las rondas puedes comprar personajes llamados campeones y formar sinergias entre ellas. Existen campeones con costos que van desde una moneda hasta cinco, siendo generalmente los más costosos los que tienen un mayor potencial. Cada campeón además de contar con una habilidad única también cuenta con estadísticas base que van acordes a su costo y rol. Por ejemplo, un personaje diseñado para resistir el daño enemigo puede contar con más puntos de vida que el que esta diseñado para hacer daño, pero también entre campeones con mismo rol pero diferente costo, suele tener mayores atributos el que cuesta más.

Problema

Riot Games frecuentemente está diseñando las próximas versiones de su juego Teamfight Tactics (TFT) cambiando campeones y mecánicas. Dado que cada campeón tiene atributos únicos, suele ser complicado ajustarlos para el costo en el que tiene que encajar o viceversa. A veces se lanza el nuevo set o versión con ciertos campeones que resultan injustamente más poderosos provocando desequilibrios que afectan la variedad de opciones y consecuentemente la jugabilidad.

Objetivo

Desarrollar una herramienta de machine learning que nos permita asignar el costo de un campeón con base en sus estadísticas base.

Solución

El ETL se realizó con el código del archivo Data_Transformation.py y los datos limpios se almacenan en el documento TFT_Champion_Transformed.csv. Esto se hizo con la intención de no ejecutar la limpieza de los datos cada que ejecutamos el archivo de machine learning. A continuación se especifica más a detalle esta etapa.

Extracción de los datos

Los datos se descargaron de Kaggle a través de la siguiente URL, compartidos por un miembro de la comunidad.

https://www.kaggle.com/datasets/gyejr95/league-of-legends-tftteamfight-tacticschampion?select=TFT_Champion_CurrentVersion.csv

Los datos son tabulares. Las columnas son principalmente numéricas, aunque también hay textos. Son 52 instancias, una por cada campeón.

```
name,cost,health,defense,attack,attack_range,speed_of_attack,dps,skill_name,skill_cost,origin,class
gangplank,5,1000,30,60,1,1.0,60,gangplank_orbitalstrike,100/175,Space Pirate,"['Mercenary', 'Demolitionist']"
graves,1,650,35,55,1,0.55,30,graves_smokegrenade,50/80,Space Pirate,['Blaster']
neeko,3,800,35,50,2,0.65,33,neeko_popblossom,75/150,Star Guardian,['Protector']
```

Transformación

Primero fue necesario eliminar las columnas que no eran relevantes o el tipo de dato no era manejable para este algoritmo. Eliminamos name, class, origin y skill_name.

En el caso de la columna skill_cost observamos que el tipo de dato es un string que contiene dos enteros separados por una diagonal. Esta columna representa con cuanta energía o maná inicia el combate y cuánta necesita el personaje para ejecutar su habilidad. Un campeón que requiera una cantidad muy alta de maná para lanzar una habilidad o que inicie el combate con muy poca cantidad, podría repercutir en qué tan poderosa es esa unidad. Por lo tanto, esta columna nos es relevante para nuestro objetivo. A partir de esta creamos dos columnas llamadas inicial_mana y skill_cost.

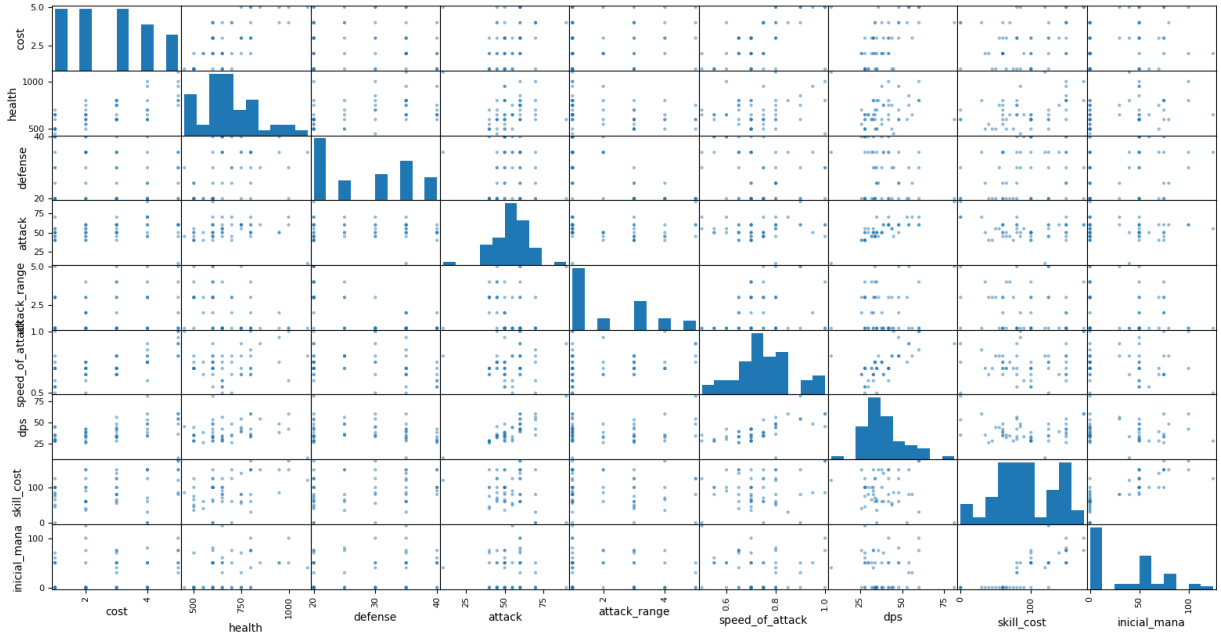
Revisando las instancias observamos que hay ciertos campeones que no utilizan maná para lanzar su habilidad. Por lo tanto, para estas unidades este atributo lo convertiremos a cero para que de esta forma solo repercuta el resto de sus estadísticas.

```
jhin,4,600,20,90,5,0.9,81,jhin_whisper,-,Dark Star,['Sniper']
jinx,4,600,20,70,3,0.75,53,jinx_getexcited,-,Rebel,['Blaster']
```

Por otra parte, nuestra columna objetivo (cost) la ponemos al final de la tabla para manejarla más fácilmente. Nos queda algo así:

	health	defense	attack	attack_range	speed_of_attack	dps	skill_cost	inicial_mana	cost
0	1000	30	60	1	1.00	60	175.0	100.0	5
1	650	35	55	1	0.55	30	80.0	50.0	1
2	800	35	50	2	0.65	33	150.0	75.0	3

Posteriormente realicé una matriz de dispersión para ver si alguna variable no mostraba una relación lineal y hacer el ajuste.



Del gráfico podemos identificar algunos patrones. Mientras en algunos hay una tendencia lineal clara (Ej. health vs cost) que hace notar que a mayor sea el costo del campeón, mayor serán sus puntos de vida, en otros pareciera una dispersión de los datos uniforme (Ej. defense vs cost), sin embargo, en el gráfico de defense vs health vemos que también a mayor defensa mayor vida lo que podría indicar que a mayor defensa también mayor el costo del personaje por lo que no es una columna que no esté aportando información.

Si observamos el conjunto de datos podemos observar un último problema. Existen atributos que pueden ser 1000 y otros que van entre 0 y 1. Al trabajar con valores grandes y potencias ocasiona desbordamientos de variables y que sea más complicado encontrar los hiper parámetros adecuados resultando en que el modelo no converja. Para solucionar este problema aplicamos un escalamiento a todas las variables para convertirlas en números entre 0 y 1. Al final obtuvimos una tabla con esta forma.

```
health,defense,attack,attack_range,speed_of_attack,dps,skill_cost,inicial_mana,cost
0.8461538461538463,0.5,0.625,0.0,1.0,0.7123287671232876,1.0,0.8,5
0.3076923076923077,0.75,0.5625,0.0,0.10000000000000009,0.3013698630136986,0.45714285714285713,0.4,1
0.5384615384615385,0.75,0.5,0.25,0.30000000000000004,0.3424657534246575,0.8571428571428571,0.6,3
```

Carga

Los datos originales utilizados se encuentran disponibles en el repositorio en el archivo TFT_Champion_CurrentVersion.csv: https://github.com/ViejoAgrio/Machine-Learning

Algoritmo

El archivo No_framework.py en la carpeta de scripts contiene el código que obtiene los parámetros "b_i" de una función para calcular el costo de cada campeón según sus estadísticas base. Esta función la obtenemos por el método de regresión lineal, utilizando como modelo la función $f(x) = b_n x_n + b_{n-1} x_{n-1} + \cdots + b_1 x_1 + b_0$ dónde $f(x)$ representa el costo del campeón y las x_i cada una de las estadísticas base de este. Para calcular el error primero utilizamos una función que convierte la evaluación de la función $f(x)$ a su entero más cercano (simulando una regresión logística múltiple) y posteriormente hacemos el promedio del error al cuadrado (MSE) entre los datos reales y la predicción del modelo. Por otra parte, para conocer la dirección en que deben ajustarse los parámetros tras cada época usamos el algoritmo de gradiente descendiente.

El entrenamiento y las pruebas se hacen por medio de validación cruzada, después de revolver las instancias, separamos subconjuntos de tamaño "x" y entrenamos con el resto, luego evaluamos con este subconjunto de "x" instancias. Repetimos hasta haber evaluado cada fila. Hacemos esto para minimizar el sesgo que podría ocasionar entrenar y probar con el mismo conjunto de datos. De esta forma evitamos caer en que el modelo solo haya "memorizado" los datos.

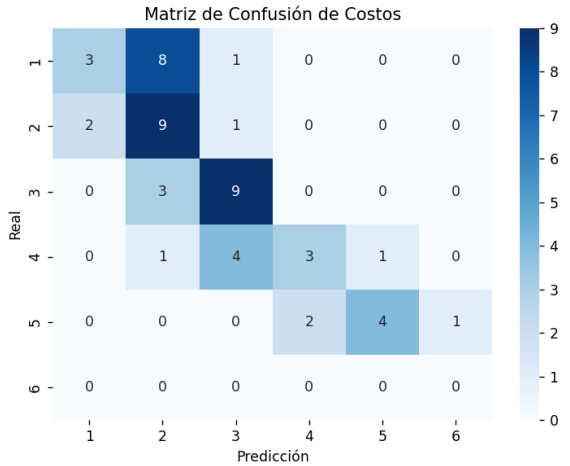
El funcionamiento del código está más especificado en los comentarios de este.

Resultados y conclusiones

Las predicciones del modelo se ven de esta forma.

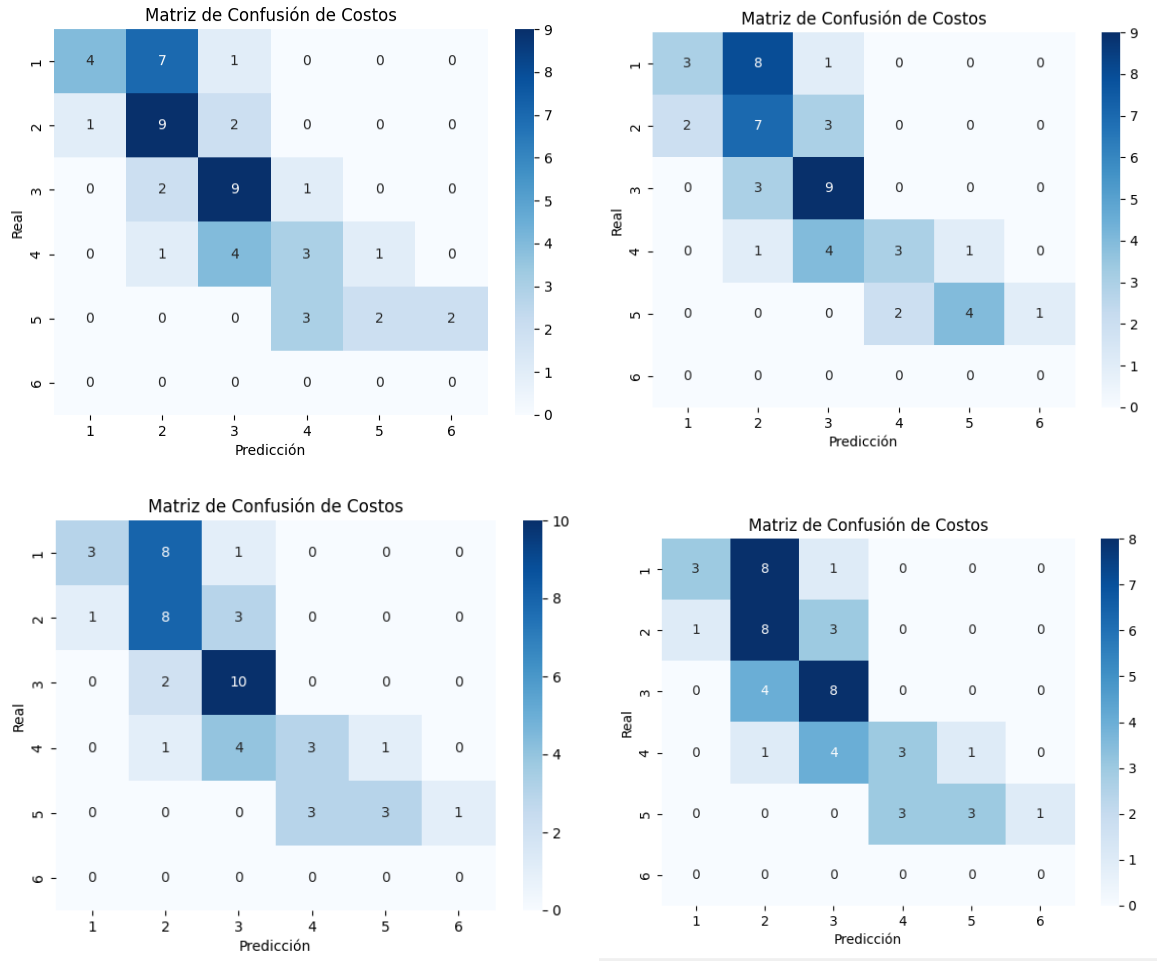
```
Predictions: [3, 4, 5, 4, 2]
Actual: [3, 5, 5, 4, 1]
Block 0: Error = 0.4000
Predictions: [3, 4, 1, 3, 2]
Actual: [1, 5, 1, 3, 1]
Block 1: Error = 1.2000
Predictions: [2, 3, 4, 3, 6]
Actual: [2, 3, 3, 3, 5]
Block 2: Error = 0.4000
Predictions: [3, 1, 3, 1, 2]
Actual: [3, 2, 2, 2, 1]
Block 3: Error = 0.8000
Predictions: [4, 2, 2, 3, 2]
Actual: [4, 1, 1, 3, 4]
Block 4: Error = 1.2000
Predictions: [1, 3, 2, 3, 3]
Actual: [1, 3, 1, 4, 4]
Block 5: Error = 0.6000
Predictions: [2, 2, 3, 4, 2]
Actual: [2, 2, 3, 5, 1]
Block 6: Error = 0.4000
Predictions: [3, 3, 3, 4, 3]
Actual: [2, 3, 4, 4, 4]
Block 7: Error = 0.6000
Predictions: [2, 2, 4, 2, 2]
Actual: [1, 3, 4, 1, 2]
Block 8: Error = 0.6000
Predictions: [5, 2, 2, 5, 2]
Actual: [5, 3, 2, 5, 2]
Block 9: Error = 0.2000
Predictions: [2, 2]
Actual: [2, 2]
Block 10: Error = 0.0000
```

El promedio de error para esta simulación fue 0.5818, lo que nos indica que generalmente una predicción está errada por la raíz de esa cantidad (0.761). En consecuencia, podemos afirmar que el modelo predice correctamente algunos costos y los erróneos varían en su mayoría por una categoría hacia arriba o abajo. Esto lo podemos ver más fácilmente en la matriz de confusión.

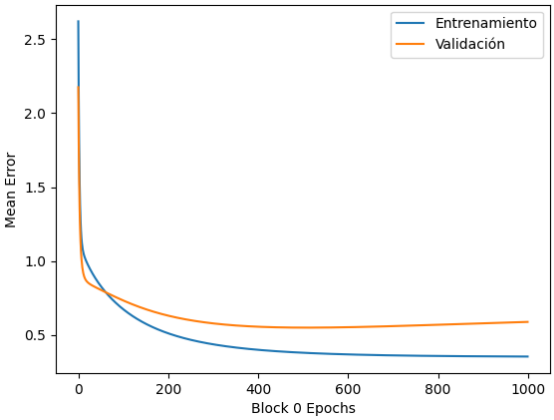


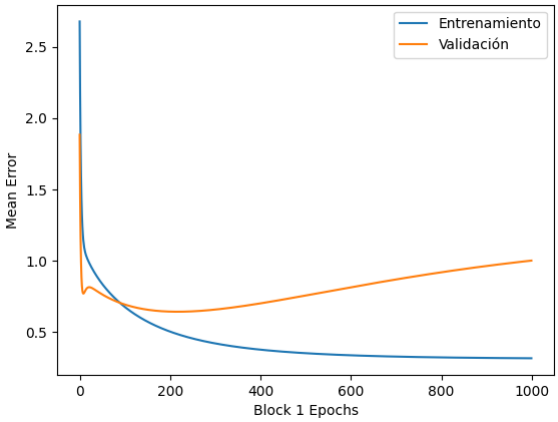
Tras múltiples ejecuciones del código encontramos que el patrón es el mismo. Suele fallar en un 75% de veces prediciendo los campeones de costo 1 asignándoles 2 y suele acertar en un 75% de las veces en el caso de predecir los de coste 3. Para los costes 4 y 5 erra en

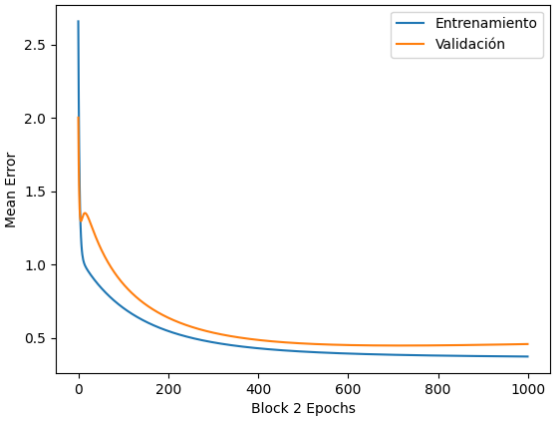
poco más del 50% de los casos y en el caso de coste 2 acierta en el 66% de veces. Generalmente acierta 27 de 52 instancias (Precisión: 51%).

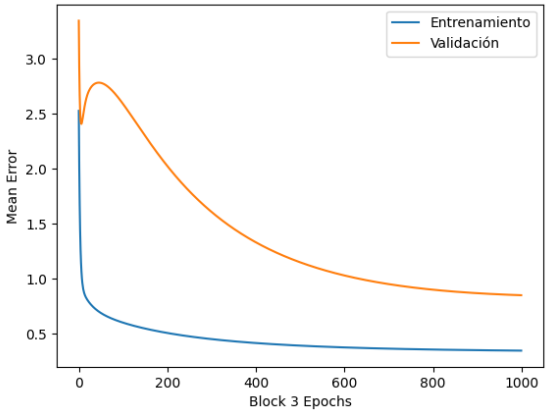


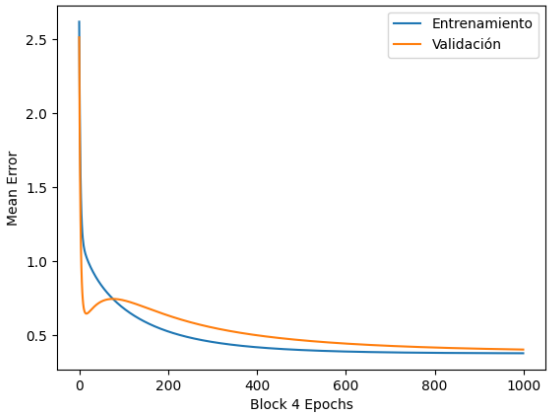
La tasa de aprendizaje utilizada fue de 0.1 y cada bloque de entrenamiento fue sometido a 1000 épocas. Estos hiper parámetros fueron seleccionados porque llegaban al límite de aprendizaje rápidamente. Al experimentar con números de épocas muy grandes el error no bajaba de 0.38 por lo que no tenía caso aumentarlas. Los gráficos de error vs época del entrenamiento y validación se ven de esta forma

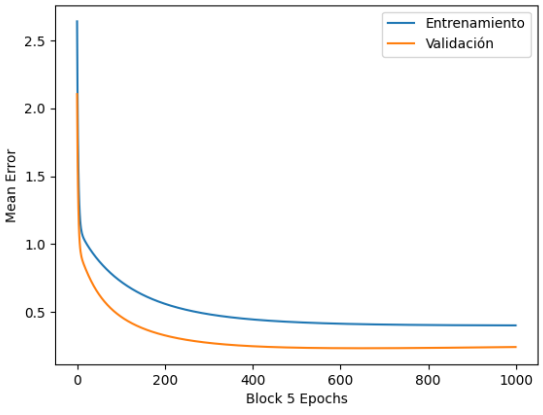

Entrenamiento
Validación
Mean Error
Block 0 Epochs

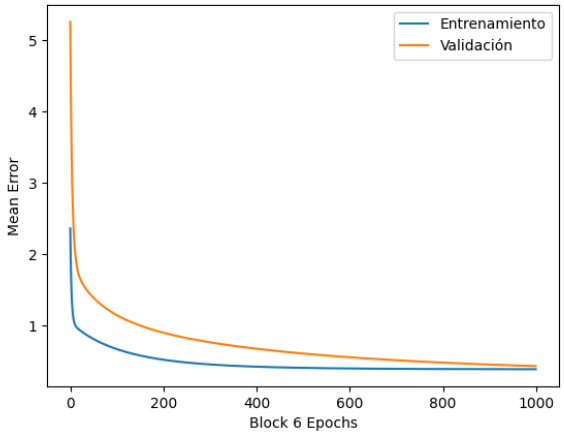

Entrenamiento
Validación
Mean Error
Block 1 Epochs

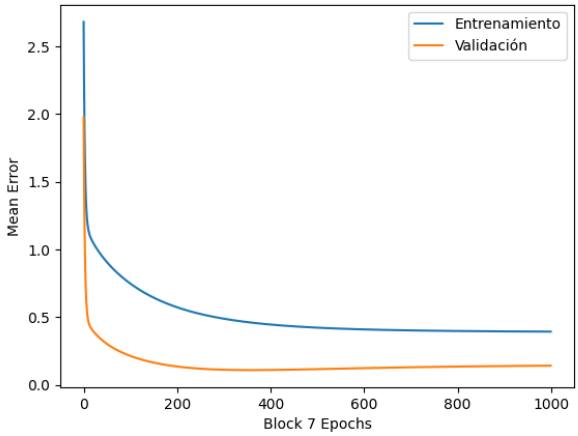

Entrenamiento
Validación
Mean Error
Block 2 Epochs

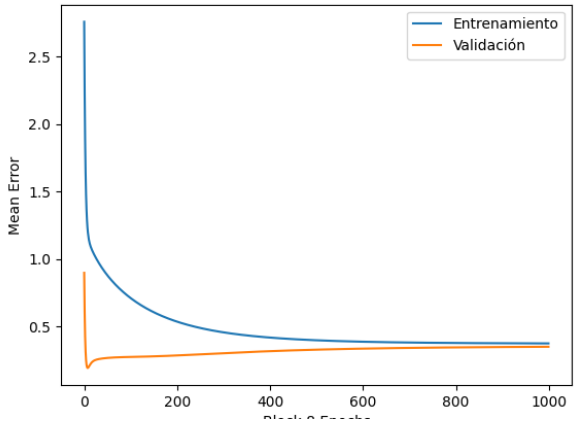

Entrenamiento
Validación
Mean Error
Block 3 Epochs

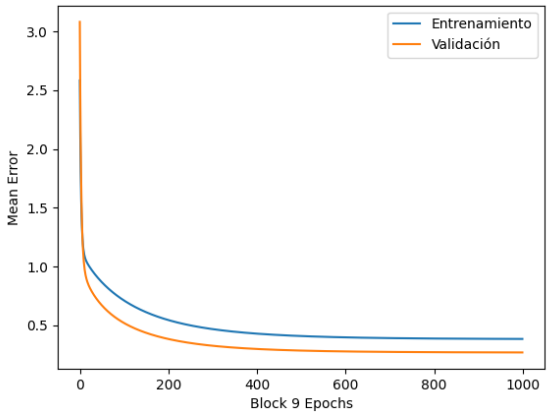

Entrenamiento
Validación
Mean Error
Block 4 Epochs

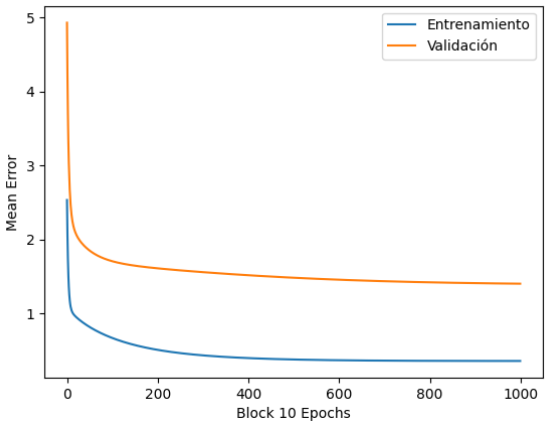

Entrenamiento
Validación
Mean Error
Block 5 Epochs







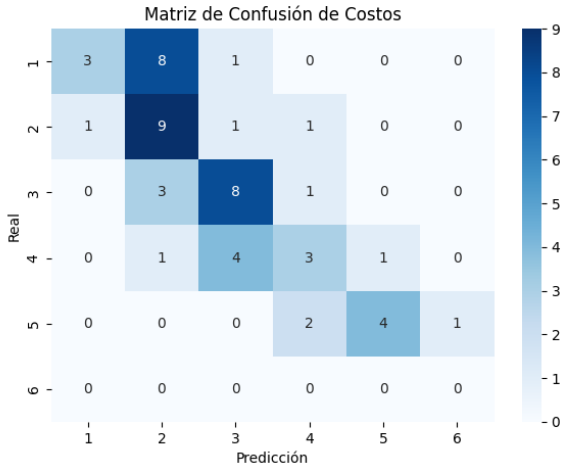




- Podemos concluir que el modelo sí aprende pues la curva de entrenamiento baja y así se mantiene.
- El comportamiento del modelo varía mucho dependiendo del bloque de datos. En algunos el error de la validación es incluso menor al de entrenamiento lo que podría indicar que el modelo generaliza bien o que ese bloque tenía ejemplos muy sencillos, mientras en otros cómo en el bloque 1 podríamos interpretar un problema de

overfitting pues el error de validación aumenta ante un caso que desconoce. Esto puede suceder debido a que ciertos bloques contuvieron campeones que no tenían un claro patrón entre sus estadísticas y sus costos, pues existen más atributos de los que depende el costo del campeón como su escalado en el tiempo y el potencial de su habilidad.

- Los casos dónde el error de validación baja y luego sube son pocos y ligeros, lo que indica que no hay un grave problema de overfitting.
- De la matriz de confusión podemos intuir que el modelo es muy simple para los datos. Mientras que campeones de costos bajos predice alto, para costos altos predice abajo y en medio es dónde tiene mayor afinidad un comportamiento no lineal, por lo que concluimos que este modelo no tiene la capacidad de ajustarse mucho más a los datos.
- Se intentó mejorar el modelo añadiendo features no lineales, por ejemplo: health, speed_of_attack, defense, attack al cubo o usando tangente hiperbólica y se ajustó un poco mejor, pero también aumentó la dispersión. La precisión fue la misma.



- Por otra parte, la columna objetivo son valores discretos, por lo que se adaptaría mejor un modelo lógico.

Cambio de dataset

Con el objetivo de poder comparar el comportamiento del modelo en entrenamiento, validación y prueba, era necesario incrementar el número de instancias del dataset. Anteriormente contábamos con 52 que eran pocas para poder tener un conjunto de entrenamiento y de prueba. Para solucionar esto unimos los datos de las versiones 14 y 15 del juego obteniendo un dataset de 125 instancias.

Extracción

Los datos fueron extraídos de las páginas web https://wiki.leagueoflegends.com/en-us/TFT:List_of_champions/Base_statistics/Set_14 y https://www.datatft.com/database#unit. Los datos originales se encuentran en los archivos TFT_set_14_raw.txt y TFT_set_15.csv.

Transformación

Para los datos de la versión 14 utilizamos un script de Python (Data_transformation_set_14.py) que nos ayudó a pasar los datos copiados de la página web a un csv con el formato que tenían los de la versión 15. Los datos de la versión 15 fueron extraídos manualmente por lo que en este proceso se hizo la discriminación de features y el acomodo de las columnas.

Posteriormente, juntamos los datos de ambas versiones en el archivo TFT_set_14_y_15.csv. Para poderlo utilizar en nuestro modelo era necesario someterlo a un escalado previamente. Para esto utilizamos el script MinMaxScaler.py. Nos quedó de esta forma:

Cabe destacar que hemos agregado nuevas columnas con el objetivo de diferenciar mejor los costos de los campeones. Ahora añadimos el escalado de vida y ataque.

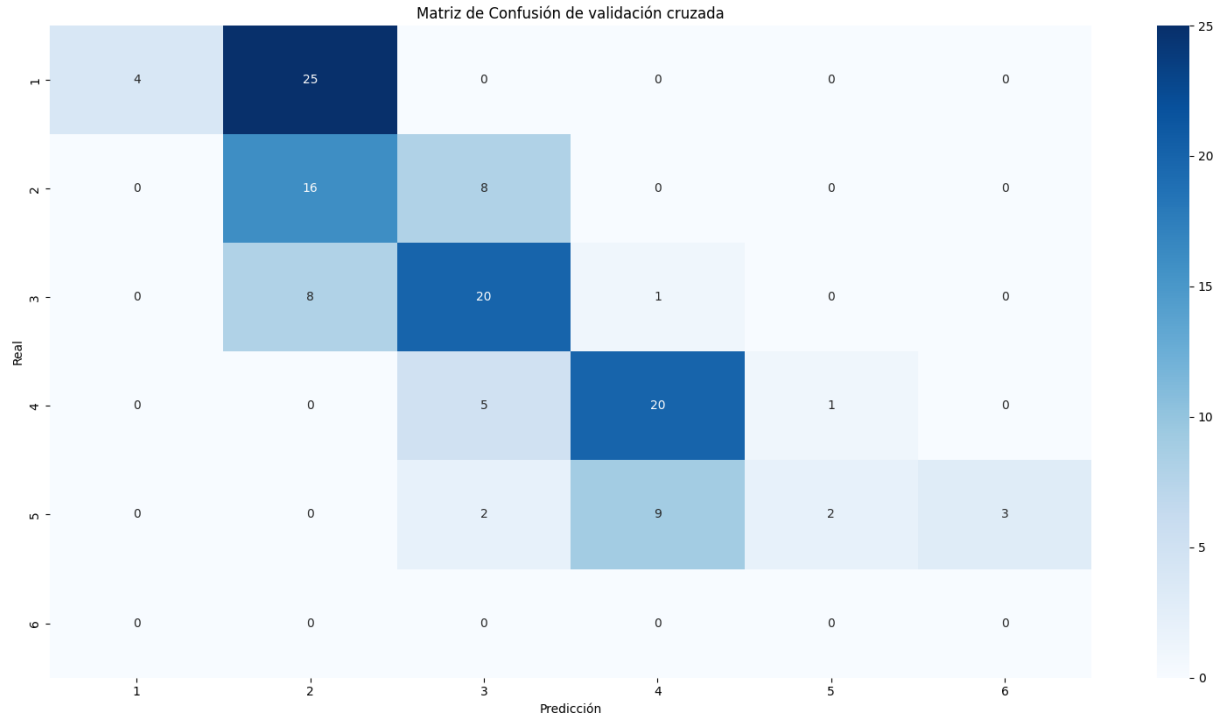
Finalmente revolvimos las instancias y tomamos 37 para pruebas y 88 para entrenamiento. Cada conjunto lo separamos en los archivos TFT_set_14_y_15_test.csv y TFT_set_14_y_15_train.csv respectivamente.

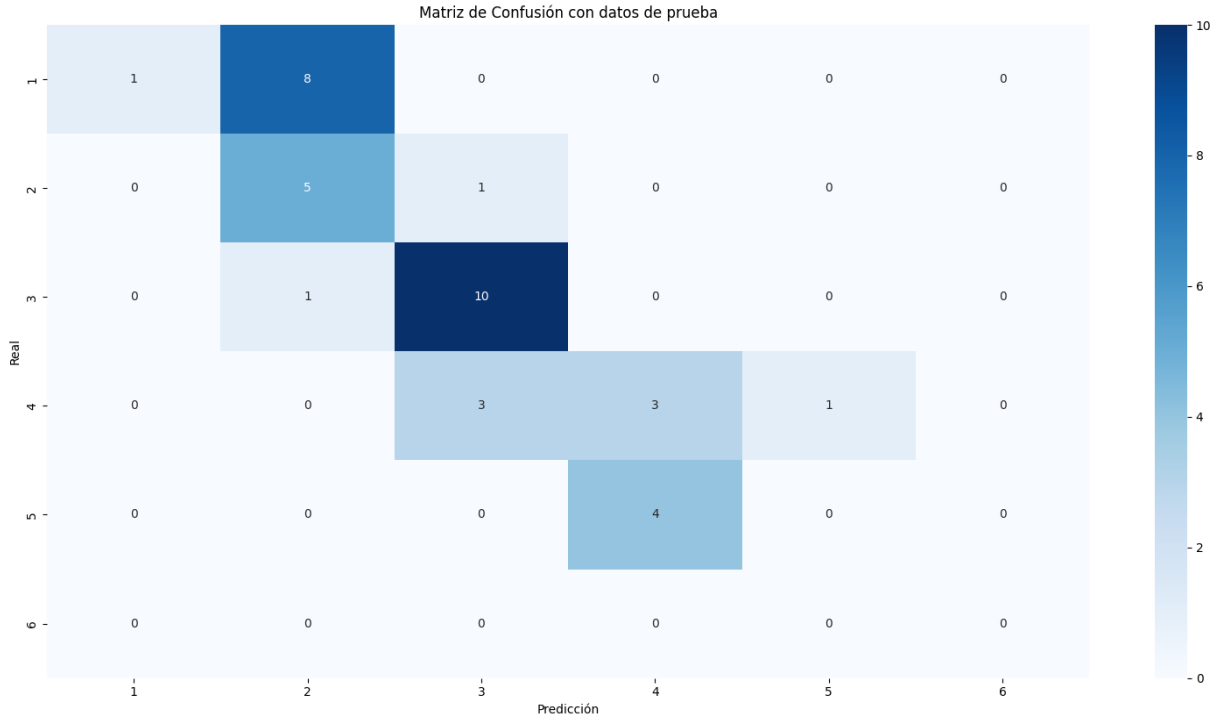
Carga

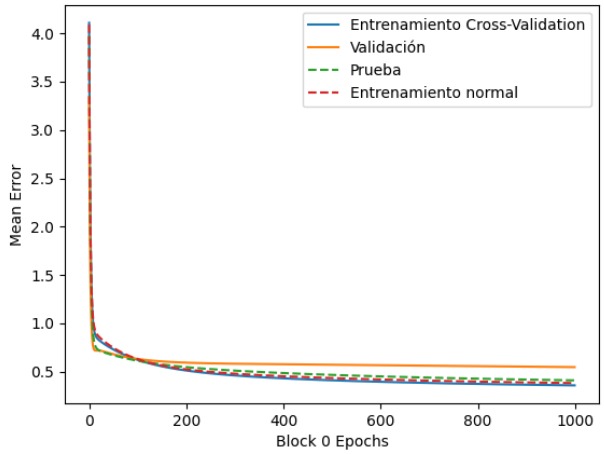
Cada conjunto de datos que se ha ido utilizando están en el repositorio https://github.com/ViejoAgrio/Machine-Learning en la carpeta de Datasets.

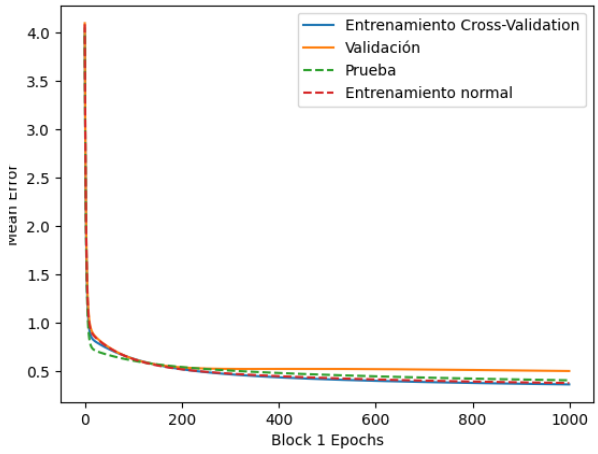
Resultados

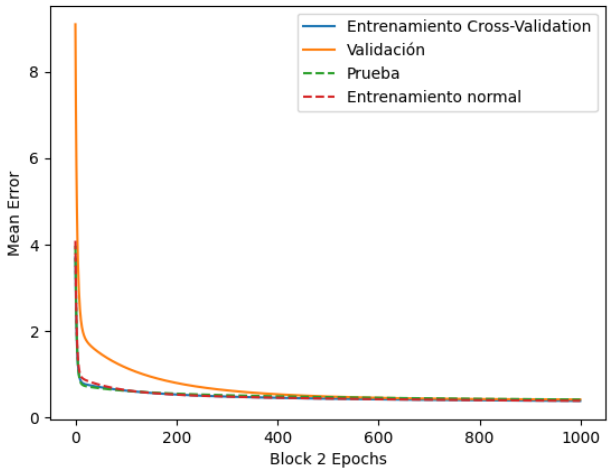
Con estos cambios podemos comparar el comportamiento del modelo en el entrenamiento, validación y pruebas. Los resultados fueron los siguientes.

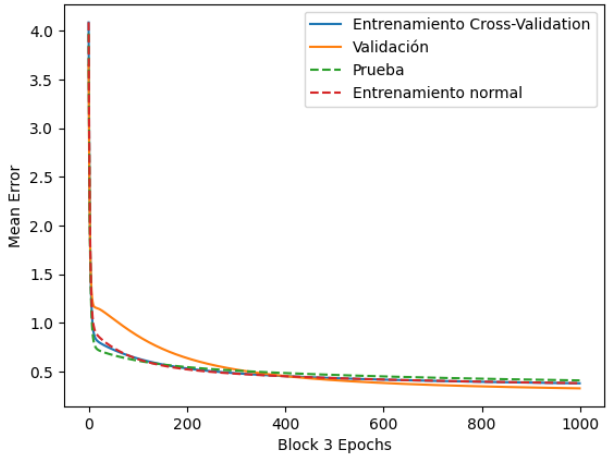

Matriz de Confusión de validación cruzada
Real
Predicción
1 2 3 4 5 6
4 25 0 0 0 0
0 16 8 0 0 0
0 8 20 1 0 0
0 0 5 20 1 0
0 0 2 9 2 3
0 0 0 0 0 0
25 20 15 10 5 0

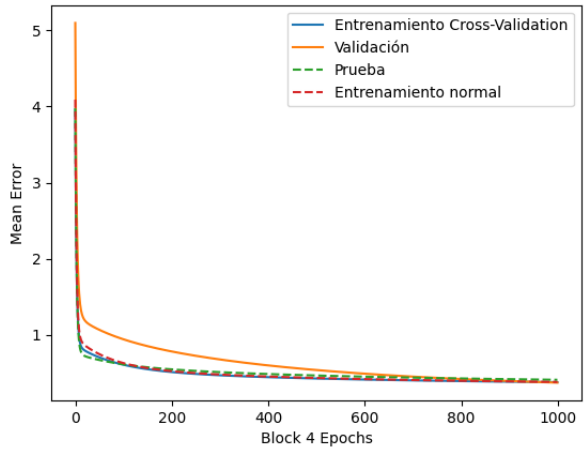

Matriz de Confusión con datos de prueba
Real
Predicción
1 2 3 4 5 6
1 8 0 0 0 0
0 5 1 0 0 0
0 1 10 0 0 0
0 0 3 3 1 0
0 0 0 4 0 0
0 0 0 0 0 0
10 8 6 4 2 0

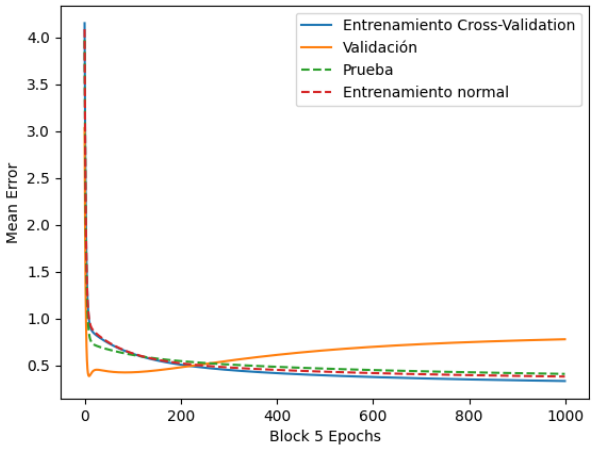

Entrenamiento Cross-Validation
Validación
Prueba
Entrenamiento normal
Mean Error
Block 0 Epochs

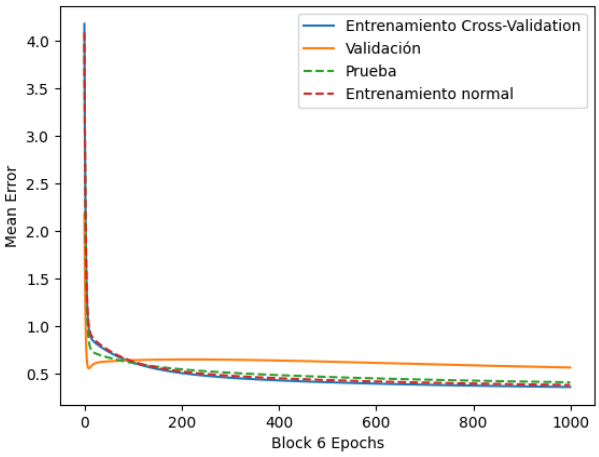

Entrenamiento Cross-Validation
Validación
Prueba
Entrenamiento normal
Block 1 Epochs

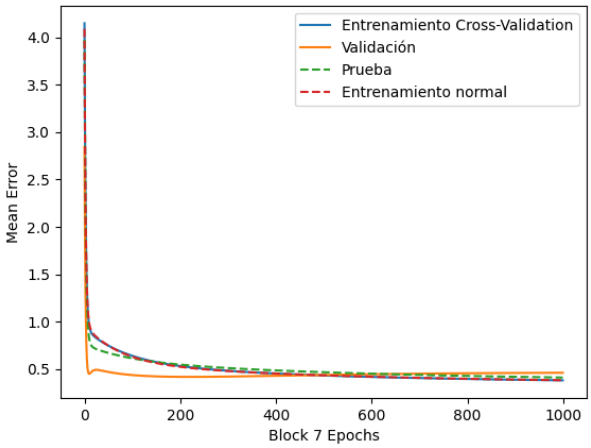

Entrenamiento Cross-Validation
Validación
Prueba
Entrenamiento normal
Mean Error
Block 2 Epochs

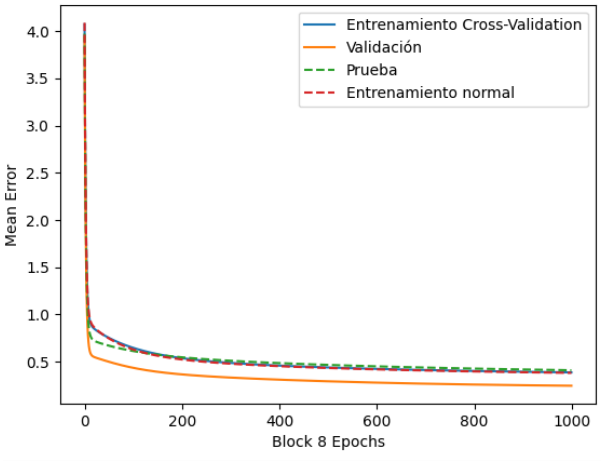

Entrenamiento Cross-Validation
Validación
Prueba
Entrenamiento normal
Mean Error
Block 3 Epochs

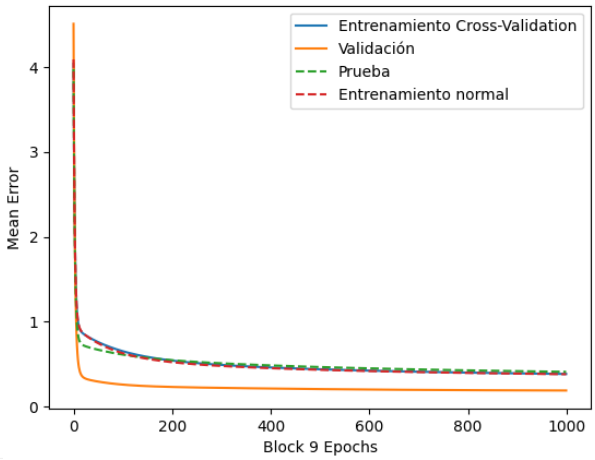

Entrenamiento Cross-Validation
Validación
Prueba
Entrenamiento normal
Mean Error
Block 4 Epochs

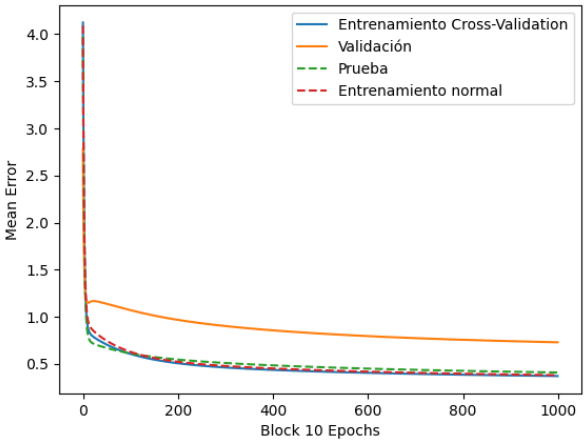

Entrenamiento Cross-Validation
Validación
Prueba
Entrenamiento normal
Mean Error
Block 5 Epochs

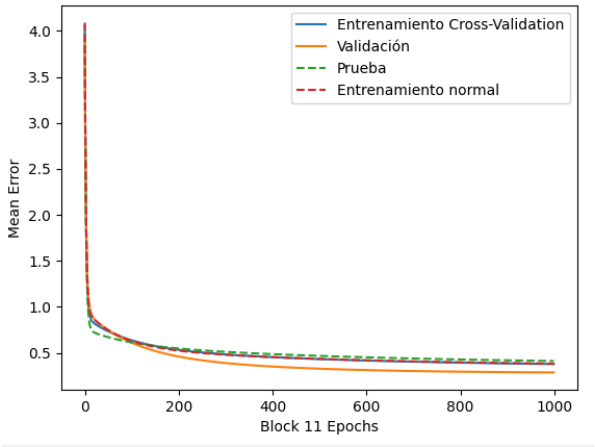












	Test	Validación
Error promedio	0.502	0.53
Precisión	51%	49%

Conclusiones

- Las curvas de error entre el entrenamiento con cross validation y el entrenamiento normal son muy similares, lo que indica que el modelo no depende de cómo se parta el dataset y consecuentemente a que no se sobre ajusta a los datos.
- A pesar del cambio de dataset a uno con más instancias y features, esto no influyó en un cambio significativo en el error promedio o precisión de las predicciones, en consecuencia, podemos intuir que es el modelo el que no se adecua para el aprendizaje de los patrones en estos datos.
- De igual forma, el patrón en las matrices de confusión se mantuvo.

Modelo con framework

Con el objetivo de encontrar un modelo que nos sea útil para diseñar y/o predecir las estadísticas base de los campeones, intentaremos utilizar otro modelo más adecuado para este tipo de datos.

Dado que nuestra variable objetivo son múltiples valores discretos y anteriormente probamos con una regresión logística múltiple, ahora intentaremos con un algoritmo basado en árboles de decisión: Random forest de clasificación con cross entropy para calcular el error.

El código de esta implementación se encuentra en el archivo With_Framework.py en la carpeta de scripts.

Marco teórico

El Random Forest es un algoritmo de aprendizaje supervisado basado en la técnica de *ensemble learning*, cuyo objetivo es mejorar la capacidad predictiva y la estabilidad de los modelos individuales. Combina múltiples árboles de decisión para generar un modelo más robusto. La idea central es que, en lugar de depender de un único árbol —que puede ser muy sensible a cambios en los datos, se construye un conjunto (o "bosque") de árboles y se toma una decisión conjunta, reduciendo la varianza y mejorando la generalización.

El proceso de construcción de un Random Forest comienza con la técnica de bagging (Bootstrap Aggregating). A partir del conjunto de datos original, se generan múltiples subconjuntos de entrenamiento mediante muestreo aleatorio con reemplazo. A cada subconjunto se le entrena un árbol de decisión independiente. Este procedimiento asegura que cada árbol vea una versión ligeramente distinta de los datos, lo cual introduce diversidad en el conjunto de modelos y evita que todos aprendan los mismos patrones.

Además del bagging, el Random Forest introduce otra fuente de aleatoriedad: en cada nodo del árbol, en lugar de evaluar todas las variables disponibles para decidir la mejor división, se selecciona al azar un subconjunto de features. Esto significa que diferentes árboles pueden tomar decisiones basadas en diferentes atributos, lo cual incrementa la diversidad entre los modelos y reduce la correlación entre ellos. Este paso es crucial para que el ensemble se beneficie de la "sabiduría de la multitud".

En la fase de predicción, el funcionamiento depende del tipo de problema. En clasificación, cada árbol vota por una clase y el Random Forest escoge la clase mayoritaria. En regresión, se calcula el promedio de las predicciones individuales.

El Random Forest presenta varias ventajas: maneja bien datos de alta dimensión, es resistente al ruido, puede capturar relaciones no lineales y proporciona medidas de importancia de variables, lo cual ayuda a interpretar los resultados. Sin embargo, también tiene limitaciones, como el hecho de que puede volverse computacionalmente costoso con grandes cantidades de árboles o datos, y que su interpretabilidad es menor que la de un único árbol de decisión.

Dataset

El dataset utilizado fue el mismo que se expone en la sección de este documento: "Cambio de dataset". La única diferencia es que para esta implementación se usó la versión sin escalamiento de los datos pues el framework lo hace en automático.

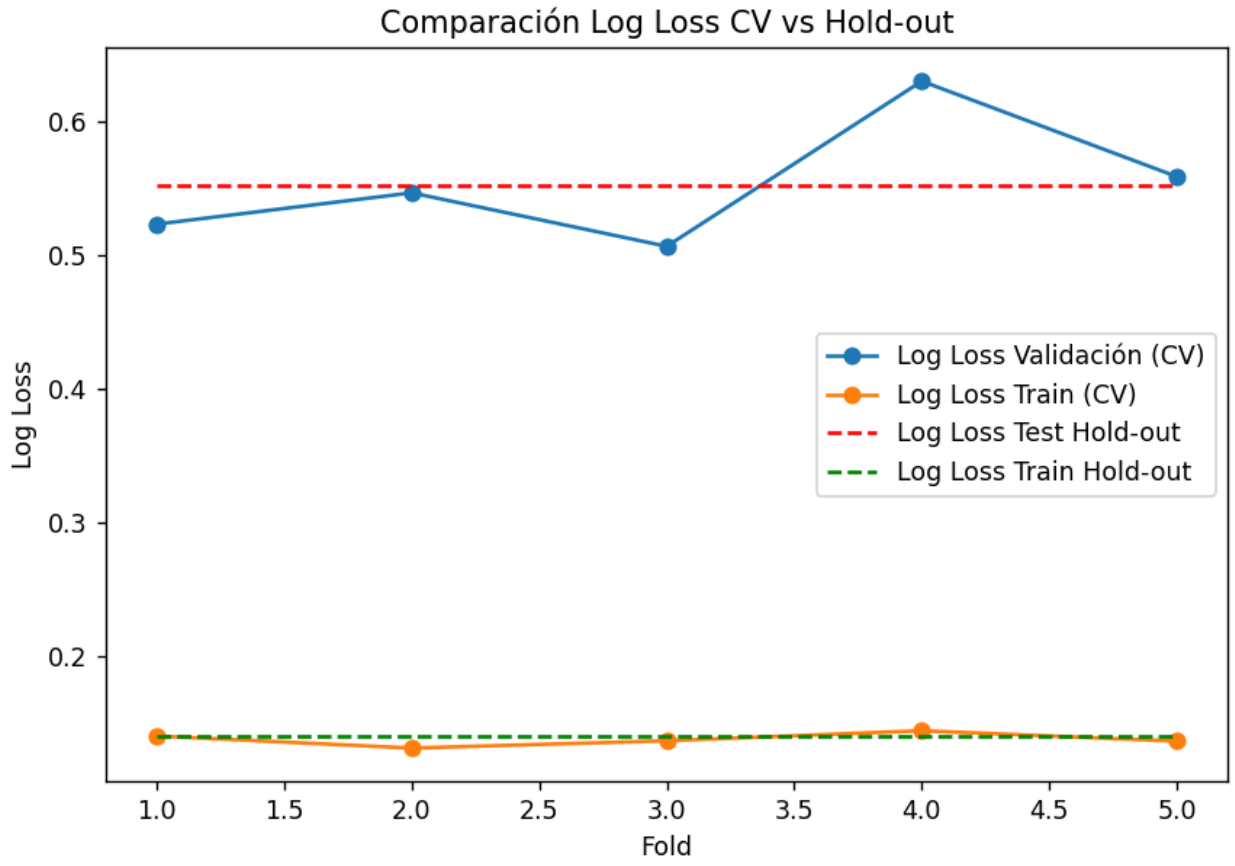
Resultados

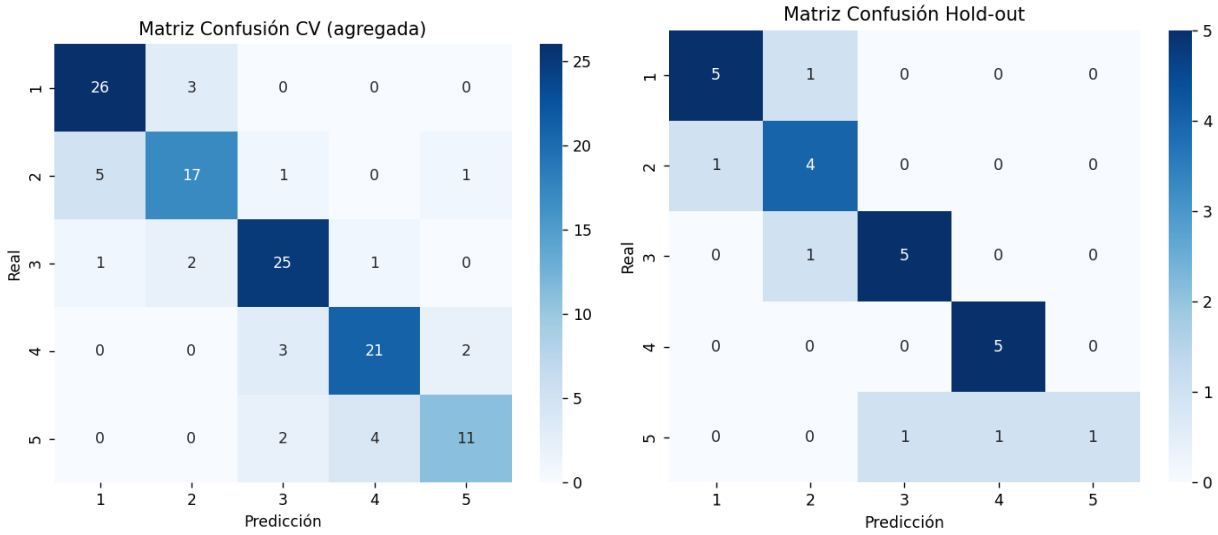
Tras el primer intento podemos ver rápidamente un gran incremento en la precisión del modelo, ahora obtenemos alrededor de un 80% tanto en validación como en test cuando con el modelo anterior era alrededor de 55%. Sin embargo, también es muy evidente un problema de overfitting dado que generalmente en el training con o sin validación cruzada obtenemos un 100% de precisión. Esto sugiere que se están memorizando los datos pues cuando lo exponemos a casos que no conoce falla significativamente más y la confianza con la que hace las aseveraciones de una clase en promedio descienden de 86% a 58% aproximadamente (Logloss = 0.14, exp(-0.14) = 0.86. Logloss = 0.54, exp(-0.54) = 0.58), es decir que para predecir el costo de un campeón generalmente esa clase tiene un 58% y entre las otras se reparten el 42% restante, por lo que normalmente hay un claro favorito para la predicción.

```
Fold 1: Train acc=1.0000 LogLoss=0.1406 | Val acc=0.6800 LogLoss=0.5235
Fold 2: Train acc=1.0000 LogLoss=0.1316 | Val acc=0.7600 LogLoss=0.5469
Fold 3: Train acc=1.0000 LogLoss=0.1370 | Val acc=0.8400 LogLoss=0.5069
Fold 4: Train acc=1.0000 LogLoss=0.1446 | Val acc=0.8800 LogLoss=0.6304
Fold 5: Train acc=1.0000 LogLoss=0.1367 | Val acc=0.8400 LogLoss=0.5590
```

Para solucionar el problema de overfitting se intentó limitar la profundidad de los árboles y aumentar su número (1000 árboles de 4 nodos de profundidad), sin embargo, aunque si acortó la diferencia entre la precisión en train y validación ligeramente (94.8% para train y 82% para validación), el logloss aumentó de forma significativa (la seguridad con la que predecía descendió de 58% a 42%), por lo que no se solucionó el problema y no tenemos un incremento importante de precisión.

```
Fold 1: Train acc=0.9500 LogLoss=0.4782 | Val acc=0.7200 LogLoss=0.7155
Fold 2: Train acc=0.9700 LogLoss=0.4296 | Val acc=0.7600 LogLoss=0.6944
Fold 3: Train acc=0.9400 LogLoss=0.4589 | Val acc=0.8000 LogLoss=0.6696
Fold 4: Train acc=0.9600 LogLoss=0.4548 | Val acc=0.8800 LogLoss=0.7675
Fold 5: Train acc=0.9300 LogLoss=0.4575 | Val acc=0.9200 LogLoss=0.7699
```

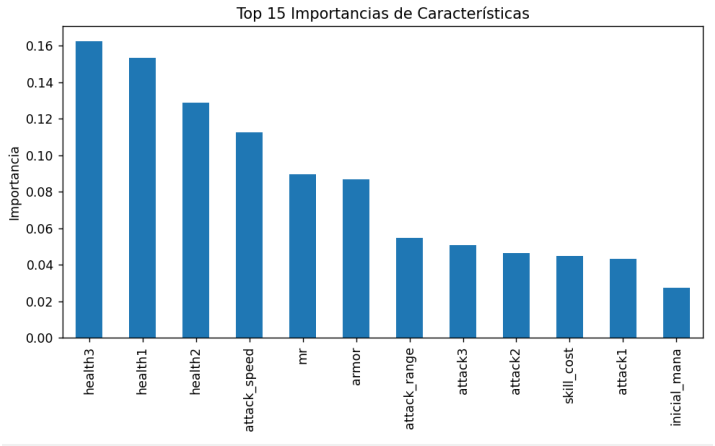




A diferencia del modelo de regresión, podemos apreciar que el patrón que se presentaba en el que fallaba con los valores extremos asignándoles un costo errado hacia el centro aquí no ocurre.

Algo que llama la atención es el caso del campeón de costo 2 que marcó como costo 5 en la matriz de confusión de validación. Este se trata de un personaje con una característica muy particular que está vinculada con su habilidad dentro del juego y en una de sus estadísticas lo hace parecer sumamente poderoso. Es un caso que vale la pena destacar pues es un gran ejemplo que ilustra el tratamiento de estas instancias ruidosas en un modelo de este tipo.

Finalmente, en la imagen de abajo podemos ver la importancia de cada uno de las features del dataset. Al parecer fue sumamente relevante añadir el escalado de vida que tenían los personajes pues cómo vemos las características más importantes para decidir el costo de un campeón fueron estas.



Conclusiones

- La hipótesis de que el modelo de random forest se ajustaría mejor que una regresión resultó ser verdadera.
- A pesar de que este modelo tiene un gran problema de overfitting, su precisión a la hora de las pruebas sigue siendo bastante superior, por lo que para resolver el problema planteado utilizaríamos este modelo.
- El uso del framework sklearn agilizó en gran medida la codificación del modelo, sin embargo, quedan muchos espacios desconocidos sobre cómo implementa las partes. ¿Cómo decide la profundidad de los árboles cuando no la indicamos? ¿Qué estrategia sigue para el armado de los árboles? ¿Las features que usa cada árbol son totalmente aleatorias? ¿Cuántas toma cada árbol? Aunque es probable que tenga opciones para ajustar cada una de estas partes, la comprensión y exploración de las diferentes técnicas son un tema que da para otro documento de investigación completo.