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.

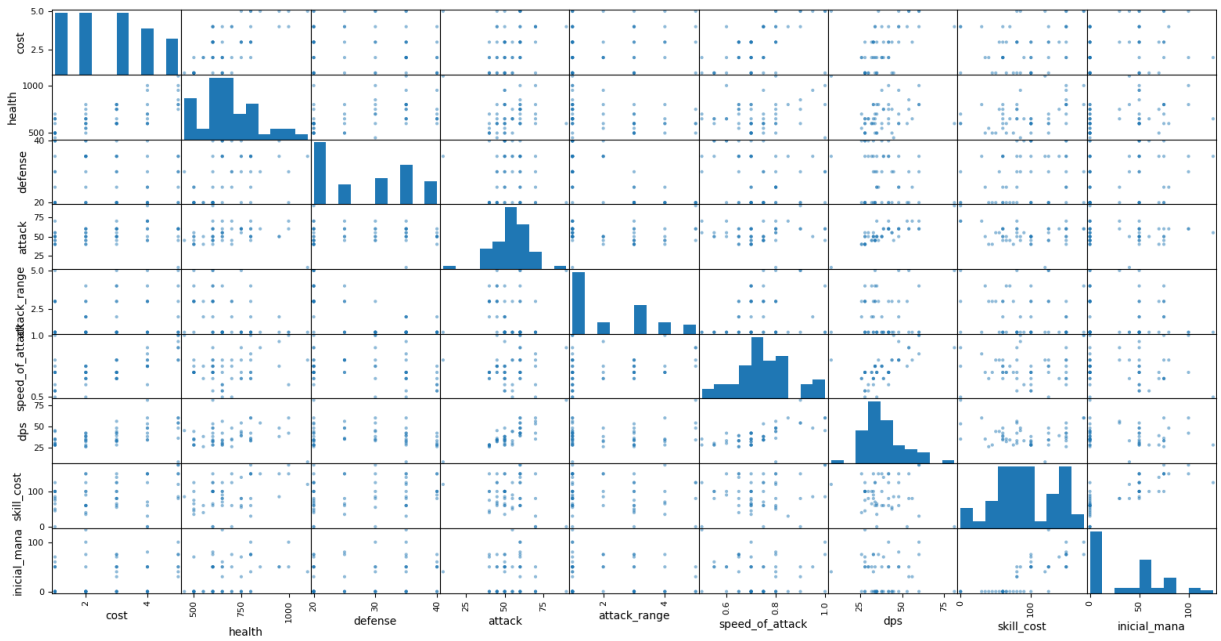
Revisando las instancias observamos que hay ciertos campeones que no utilizan maná para lanzar su habilidad. Por lo tanto, para estas unidades este atributo lo convertiremos a cero para que de esta forma solo repercuta el resto de sus estadísticas.

```
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 MachineLearning.py en esta misma carpeta 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 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.

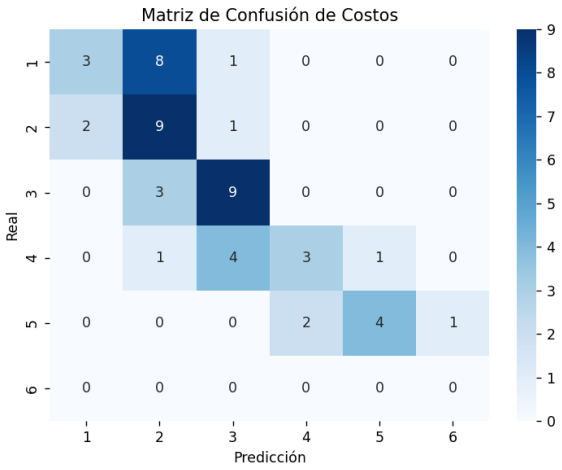
El funcionamiento del código está más especificado en los comentarios de este.

Resultados y conclusiones

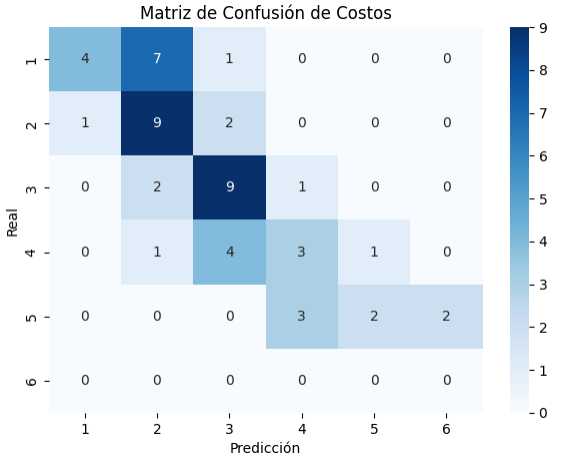
Las predicciones del modelo se ven de esta forma.

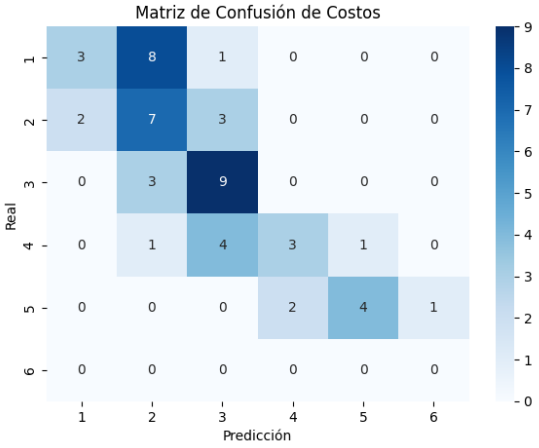
```
Predictions: [3, 4, 5, 4, 2]
Actual: [3, 5, 5, 4, 1]
Block 0: Error = 0.4000
Predictions: [3, 4, 1, 3, 2]
Actual: [1, 5, 1, 3, 1]
Block 1: Error = 1.2000
Predictions: [2, 3, 4, 3, 6]
Actual: [2, 3, 3, 3, 5]
Block 2: Error = 0.4000
Predictions: [3, 1, 3, 1, 2]
Actual: [3, 2, 2, 2, 1]
Block 3: Error = 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
```

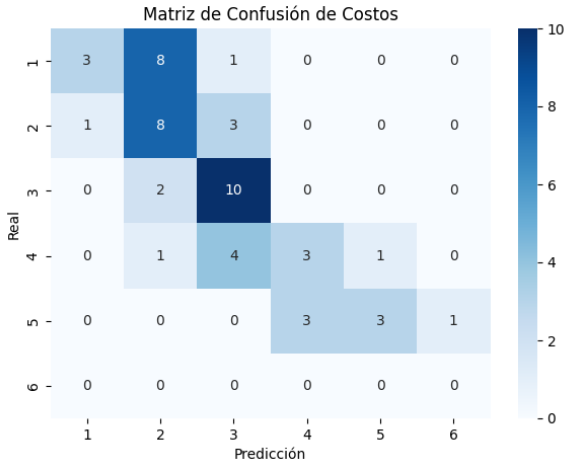
El promedio de error para esta simulación fue 0.5818, lo que nos indica que generalmente una predicción está errada por la raíz de esa cantidad (0.761). En consecuencia, podemos afirmar que el modelo predice correctamente algunos costos y los erróneos varían en su mayoría por una categoría hacia arriba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

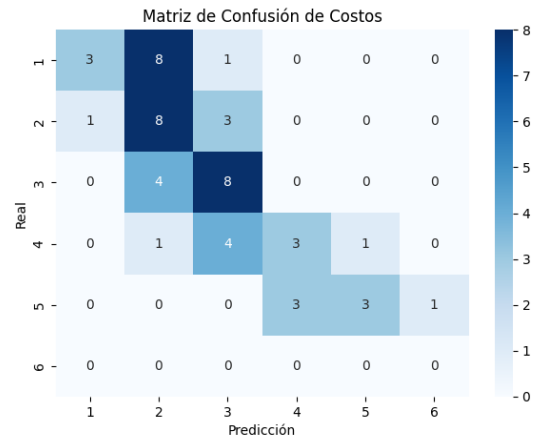


Tras múltiples ejecuciones del código encontramos que el patrón es el mismo. Suele fallar en un 75% de veces prediciendo los campeones de costo 1 asignándoles 2 y suele acertar en un 75% de las veces en el caso de predecir los de coste 3. Para los costes 4 y 5 erra en poco más del 50% de los casos y en el caso de coste 2 acierta en el 66% de veces. Generalmente acierta 27 de 52 instancias.

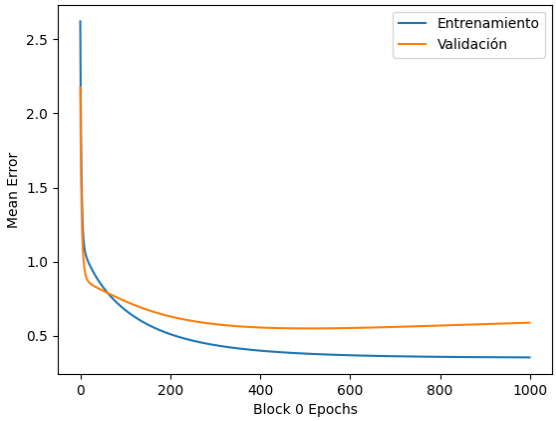


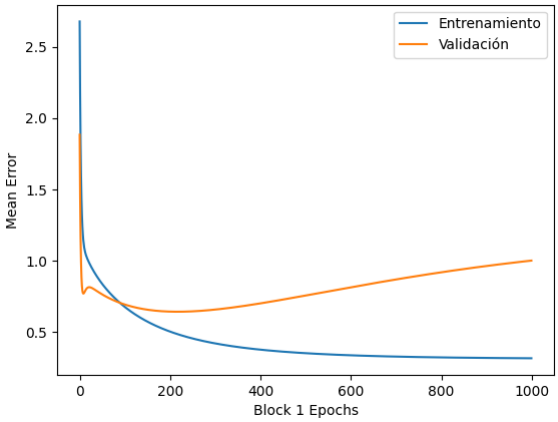


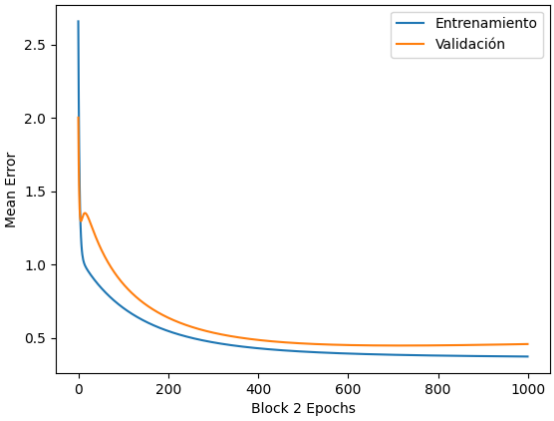


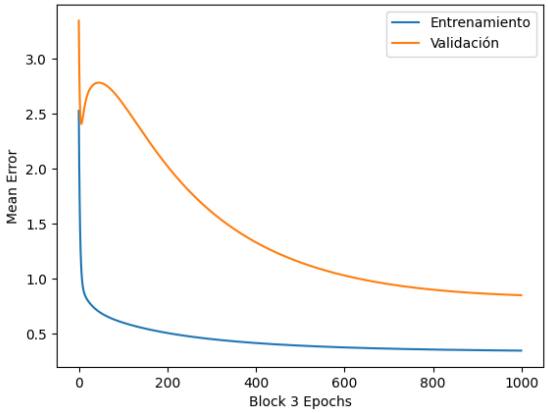


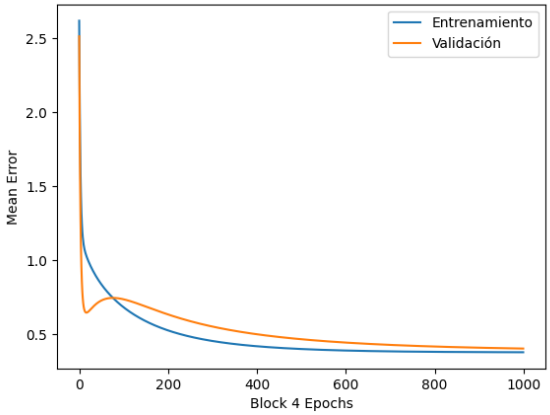
La tasa de aprendizaje utilizada fue de 0.1 y cada bloque de entrenamiento fue sometido a 1000 épocas. Estos hiper parámetros fueron seleccionados porque llegaban al límite de aprendizaje rápidamente. Al experimentar con números de épocas muy grandes el error no bajaba de 0.38 por lo que no tenía caso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

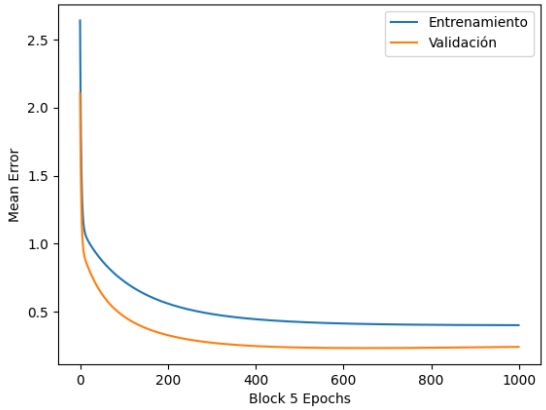


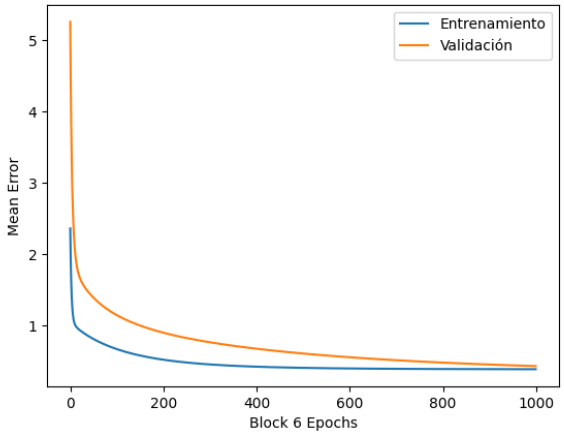


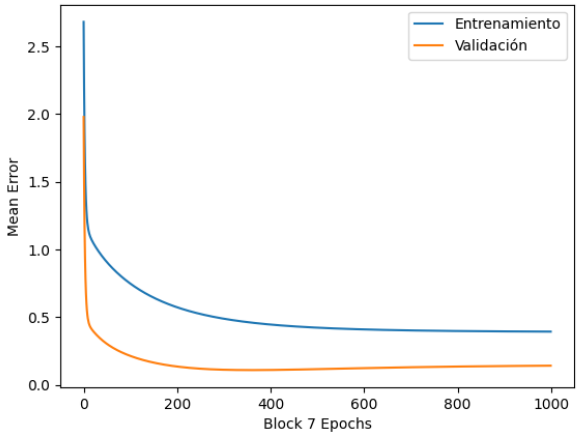


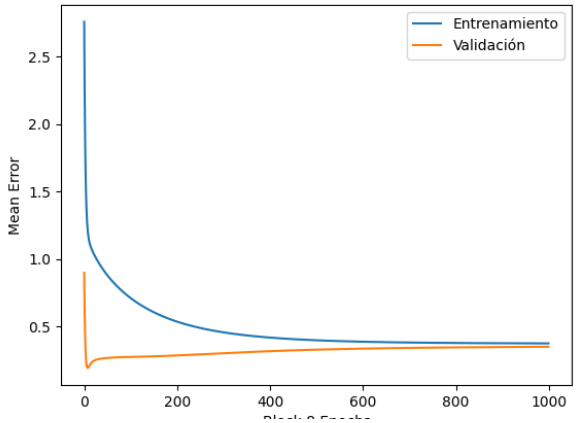


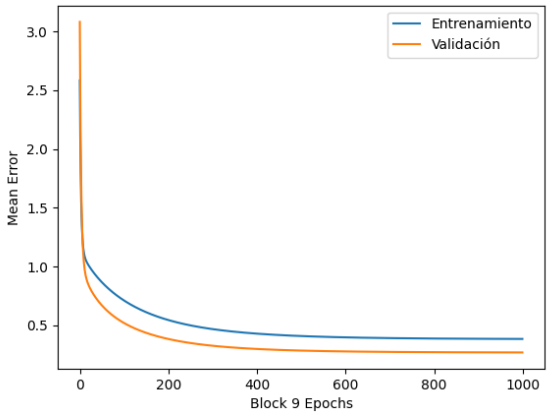


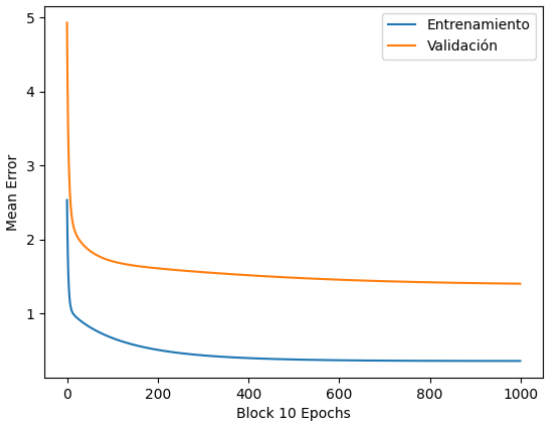












- Podemos concluir que el modelo sí aprende pues la curva de entrenamiento baja y así se mantiene.
- El comportamiento del modelo varía mucho dependiendo del bloque de datos. En algunos el error de la validación es incluso menor al de entrenamiento lo que podría indicar que el modelo generaliza bien 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 que 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 grave problema de overfitting.
- No hay underfitting pues ambas líneas se quedan abajo en la mayoría de los casos.