Universidad Complutense de Madrid
Septiembre 2020

Construcción de dataset y predicción de resultados en League of Legends

Author: Diego Candela Salomón

Máster en Big Data y Business Analytics

UCM Facultad de Ciencias Económicas y Empresariales

Abstract

In this project we use a variety of machine learning algorithms to predict the match outcomes of the online multiplayer game: League of Legends by Riot Games. Features are taken from the data that Riot Games API offers. One-hot encoding is used to transform mostly categorical data of pre-game and in-game features into a format to be used in machine learning algorithms such as: avNNet, K-nearest neighbors, Naïve-Bayes, Random Forest and XGBoost. It is found that pre-game features by themselves are weak predictors but using both pre-game and in-game features, all of our models become strong classifiers.

Palabras Clave: Machine Learning – Python – R – Neural Networks – Gradient Boosting – Decision Trees – League of Legends

Índice

Índice de figuras

Índice de cuadros

Índice de cuadros

1 Introducción

2 Introducción

El 10 de noviembre de 2019 tuvo lugar la final del Campeonato Mundial de League of Legends con un pico de 44 millones de espectadores repartidos por el mundo entero [cita]. Los e-sports se han convertido en una de las formas de entretenimiento más seguidas con League of Legends a la cabeza. Es un juego que acoge más de 100 milliones de jugadores al mes y cuyos torneos amasan millones de dólares. Con una comunidad tan grande, y con la relevancia que tiene ser un juego tan popular, el aplicar técnicas de machine learning y Deep learning para ser capaz de generar modelos de predicción que ayuden a mejorar la capacidad de comprensión del juego a equipos, entrenadores e incluso aficionados, se ha convertido en un tema de investigación importante.

2.1 Descripción del juego

"League of Legends" es un juego de estrategia por equipos creado por Riot Games publicado en 2009 en el que dos equipos de cinco campeones se enfrentan con el objetivo de derribar la base del oponente [1]. Los campeones son avatares que los jugadores escogen para representarlos durante la partida. En el momento de la redacción de este documento existen un total de 150 campeones. A medida que transcurre la partida los campeones adquieren recursos económicos y experiencia para equiparse y volverse más poderosos. Estos dos factores son esenciales para superar al equipo enemigo y destruir su base. Tanto el medio como los objetivos experimentales pasan por entender el sistema competitivo inherente al juego. League of Legends se estructura en un sistema de ligas competitivas. Los jugadores que ganan partidas consecutivas acumulan puntos que les permiten subir de liga. De igual manera, cuando se pierde una partida, se pierden puntos y se puede descender de liga. Las ligas se ordenan de la siguiente manera: bronce, plata, oro, platino, diamante, "máster" y "challenger". Los jugadores más habilidosos acceden a las ligas más altas como "máster.º çhallenger". La base de datos que se ha construido para el análisis de este documento proviene de partidas jugadas únicamente en la liga "máster".

2.2 Machine Learning y minería de datos

Podemos definir la minería de datos como la ciencia que extrae información útil de grandes cantidades de datos o bases de datos [2]. Comparte con el Machine Learning el uso de recursos computacionales para lograr dicho objetivo. Lo que entendemos como Machine Learning comprende una serie de algoritmos capaces de aprender basándose en conjuntos de datos generando así un modelo. Estos modelos se pueden aplicar a posteriori en otros conjuntos de datos para realizar predicciones, tareas de clasificación o interpretar características del conjunto de datos.

Podemos dividir los procesos de Machine Learning en aprendizaje supervisado y no supervisado. A lo largo de este documento nos centraremos en aplicaciones de algoritmos basados en aprendizaje supervisado. Esto quiere decir que cada conjunto de datos de entrenamiento cuenta con parte de los resultados o etiquetas. En este caso en concreto, la etiqueta corresponde con al victoria o derrota del equipo que juega la partida. Así, cuando generamos un modelo lo que estamos haciendo es ajustar la predicción del modelo tratando de acercarla lo más posible a la respuesta correcta. Las etiquetas que veremos en el conjunto de datos se pueden entender como victoria/derrota o 0 y 1. Estamos, por tanto, ante un problema de clasificación y entrenaremos algoritmos como: Naive Bayes, K-nearest neighbors (KNN), redes neuronales (método Avnnet), Random Forest y una variación del gradient boosting (XGBoost).

2.3 Estado del arte

Dada la popularidad de League of Legends, existen una serie de aplicaciones que registran y analizan datos en distintas modalidades: para mantener un historial personal, analizar a otros jugadores, o para rastrear las estadísticas de campeones específicos. Algunas de estas serían: League of Graphs, MetaSrc, Mobaliytics o Blitz. [3]

Se encuentran aplicaciones de estas técnicas que utilizan algoritmos de predicción en intervalos específicos de una partida en videojuegos del mismo género como DOTA 2 [4]. Posteriormente, se han desarrollado modelos basados en regresiones logísticas y random forests para League of Legends obteniendo en torno al 75 % de precisión. [5]

Existen propuestas de sistemas de predicción orientadas a los niveles más altos de la competición del juego: Riot Games, la empresa fundadora, implementa en algunos eventos competitivos las funciones de un robot predictivo, Colonel KI [6]. Colonel KI es capaz de predecir el resultado de la partida en base a distintas estadísticas y características del desarrollo previo y durante la partida. No obstante, los datos de los que aprende Colonel KI se limitan a partidas a un nivel profesional que no se traducen al día a día del juego.

3 Motivación y objetivos

3.1 Motivación

Propuestas como la de Colonel KI, se centran en datos al más alto nivel competitivo del videojuego. Naturalmente, el transcurso de una partida profesional que se desarrolla bajo la supervisión de un entrenador que dispone los medios para que un equipo entrenado de cinco jugadores ejecute una serie de estrategias está lejos del desempeño del día a día del jugador promedio. Por esta misma razón, modelos como Colonel KI no se pueden emplear fuera de los círculos competitivos y no tiene buenos resultados cuando se aplica a conjuntos de datos de jugadores de ligas inferiores.

Desarrollar modelos orientados a ligas más bajas puede ser de gran ayuda a jugadores de base que conforman la gran mayoría del público de League of Legends, a diferencia de la escena competitiva.

3.2 Objetivos

El propósito de este documento consiste en:

1. Extraer una base de datos a partir de la API pública de Riot Games suficientemente grande como para generar modelos predictivos.
2. Generar modelos de clasificación eficaces basándonos únicamente en datos que se pueden extraer previo al comienzo de la partida.
3. Generar modelos de clasificación optimizados con datos del total de la partida.

4 Metodología

4.1 Entorno de programación

Para la extracción de datos se ha utilizado Python (versión 3.7.4) dentro de la distribución de Anaconda 3 (versión 2019.10). Anaconda es una de las distribuciones de Python recomendadas para Data Science. Para el análisis de datos se ha empleado R (versión 3.6.1). R es un potente lenguaje para análisis de datos que dispone de muchos paquetes de machine learning.

4.2 Tratamiento de los datos

Los datos se pueden obtener a través de la API oficial de Riot Games. Para obtenerlos se ha utilizado el wrapper "Cassiopeia" [7] en Python. Cassiopeia es un framework diseñado para interactuar y captar datos de la API de Riot. Para construir la base de datos se ha de consultar a través de los IDs de cada partida. Con el objetivo de conseguir una lista de IDs de partidas suficientes hemos recopilado las 1000 primeras partidas del historial de jugadores pertenecientes a la liga "masters". Los datos de todo lo ocurrido antes y durante la partida se almacenan en un JSON. Esta es la estructura de datos que se obtiene al consultar una partida en la API de la que hemos de extraer las características relevantes para el análisis. Una vez hemos descargado el conjunto de datos, consistía en un total de 49769 filas y 42 columnas. Tras eliminar las filas corruptas y las repetidas, el dataset final consiste en 43452 filas. De las filas restantes, no se encuentran missings. La variable objetivo es la victoria del equipo azul: la variable toma el valor 1 si el equipo azul alcanza la victoria y 0 cuando la victoria es del equipo rojo. Esto quiere decir que estamos ante un problema de clasificación. Dada la naturaleza de los datos, se ha tomado una aproximación de "one hot encoding". Y cada columna se ha convertido en un conjunto de variables dummy, por lo que la base de datos pasó de 42 columnas a 3085.

4.3 Selección de variables

4.3.1 Datos pre-game

Uno de los objetivos principales de este trabajo es ser capaz de generar un modelo capaz de predecir eficientemente el resultado de la partida antes de que esta ocurra. Para ello, se utilizan varias de las variables de la partida que se obtienen antes de que dé comienzo:

- Selección de campeones o "picks": cada uno de los 10 jugadores ha de escoger un avatar que lo representa en la partida. Existen un total de 150 campeones en el momento en el que se redactó este trabajo. Cada campeón tiene habilidades y estadísticas distintas.
- Denegaciones o "bans": cada jugador puede denegar a ambos equipos la posibilidad de jugar con un campeón.
- Grado de maestría o experiencia: Cassiopeia nos permite hacer consultas al servidor sobre el perfil de los jugadores que participan en la partida. A medida que un jugador acumula partidas jugadas con un campeón, también se acumula experiencia. Se interpreta que este grado de experiencia ha de ser una variable con la que poder mejorar la precisión de un modelo. Es una variable continua con un rango de valores de 0 a 8732766. Sin tratarla previamente, es una variable que no aporta mucha información y es necesario transformarla. Se utilizó el paquete rpart"para tramificar la variable en función de la etiqueta y así conseguir reducirla a una variable factor con 10 niveles.

4.3.2 Datos in-game

Riot registra la información de todo lo que ocurre durante el transcurso de la partida. Atendiendo a los resultados de trabajos previos y tratando de realizar las consultas al servidor de la manera más eficiente, se escogieron los siguientes datos:

- "Primera sangre": el primer jugador en conseguir la primera eliminación de un oponente obtiene un bonus extra de oro. Toma el valor 1 si el equipo azul lo logra y 0 si no.

- "Primer dragón": el equipo que consiga hacerse con el primer dragón de la partida obtiene una ventaja estratégica. Toma el valor 1 si el equipo azul lo logra y 0 si no.
- "Eliminaciones de dragones": la cantidad de dragones que consigue eliminar el equipo azul. El rango de la variable es de 0 a 5.
- "Primer heraldo": el equipo que consiga hacerse con el primer heraldo obtiene una ventaja estratégica. Toma el valor 1 si el equipo azul lo logra y 0 si no.
- "Eliminaciones de heraldo": la cantidad de heraldos que consigue eliminar el equipo azul. El rango de la variable es de 0 a 3.
- "Primera torre": el equipo que consigue derribar la primera torre de la partida al oponente obtiene una ventaja estratégica y un bonus de oro. Cuantas más torres se derriban más cerca se está de la victoria. Toma el valor 1 si el equipo azul lo logra y 0 si no.
- "Eliminaciones de torres": la cantidad de torres que consigue eliminar el equipo azul. El rango de la variable es de 0 a 12.
- "Primer inhibidor": el equipo que consigue derribar el primer inhibidor de la partida al oponente obtiene una ventaja estratégica. Cuantos más inhibidores se derriban más cerca se está de la victoria. Toma el valor 1 si el equipo azul lo logra y 0 si no.

5 Resultados

5.1 Algoritmos y resultados

5.1.1 Avnnet

Avnnet forma parte del paquete caret [8]. Consiste en un modelo donde se entrenan varias iteraciones de la misma red neuronal utilizando distintas seeds aleatorias. En los problemas de clasificación la puntuación de los modelos se promedian y a partir de ahí se traducen en la predicción de las clases. [fuente: documentación caret].

Los resultados del modelo entrenado con el dataset previo al comienzo de la partida devuelven una precisión cercana al azar: 0.559. Kappa: 0.118. Los parámetros finales son size = 5, decay = 0.01, bag = False y RMSE = 0.65983.

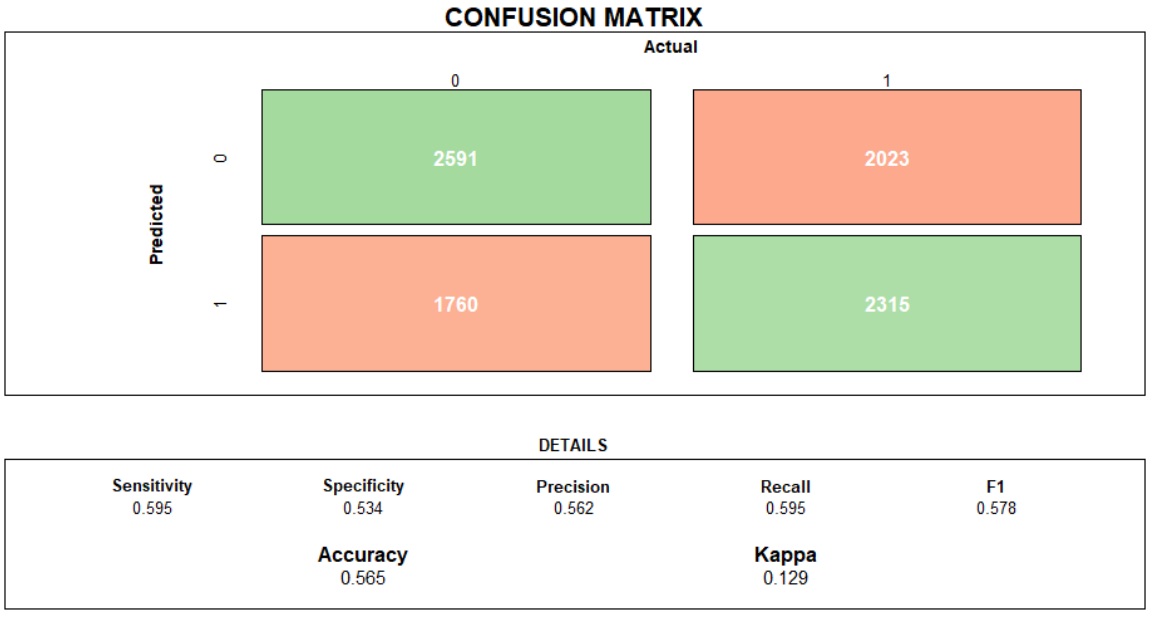

Figura 5.1: Matriz de confusión. Avnnet pre-game.

Los resultados cambian drásticamente cuando entrenamos el modelo con el dataset al completo. Con los parámetros size = 5, decay = 0.01, bag = False y RMSE = 0.3058 se obtiene una precisión = 0.906

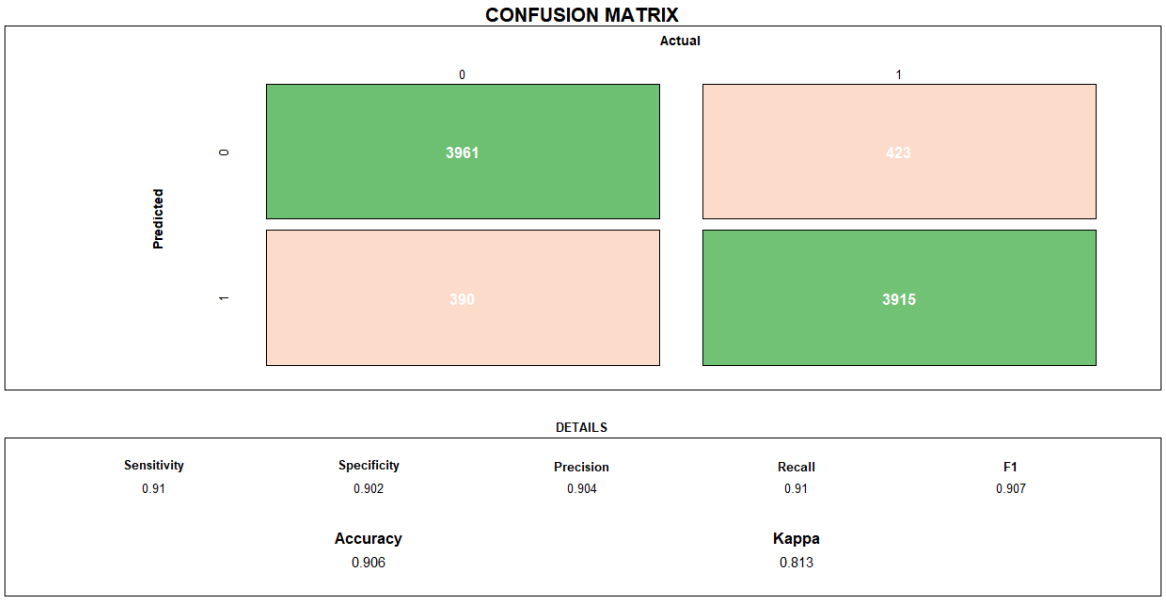

Figura 5.2: Matriz de confusión. Avnnet completo.

5.1.2 Naïve Bayes

El principio del algoritmo Naïve Bayes se basa en el cálculo de P(Y) utilizando la fórmula de Bayes para calcular la probabilidad posterior P(Y/X) (la probabilidad de que un objeto pertenezca a cierta clase) y así seleccionar la clase con la mayor probabilidad de pertenencia. Las modalidades más conocidas del algoritmo incluyen: Naïve Bayes Gaussiana, Naïve Bayes de Bernouilli y la Distribución Polinomial de Naïve Bayes. El uso de cada uno de estos métodos depende de la clase de problema para la que se estén usando. Puesto que en este caso nos encontramos ante un problema de clasificación, emplearemos el algoritmo de Naive Bayes de Bernouilli.

Los resultados del modelo entrenado con el dataset previo al comienzo de la partida se mantienen próximos al azar con una precisión = 0.549, fL = 0, adjust = 1, usekernel = False y RMSE = 0.67184

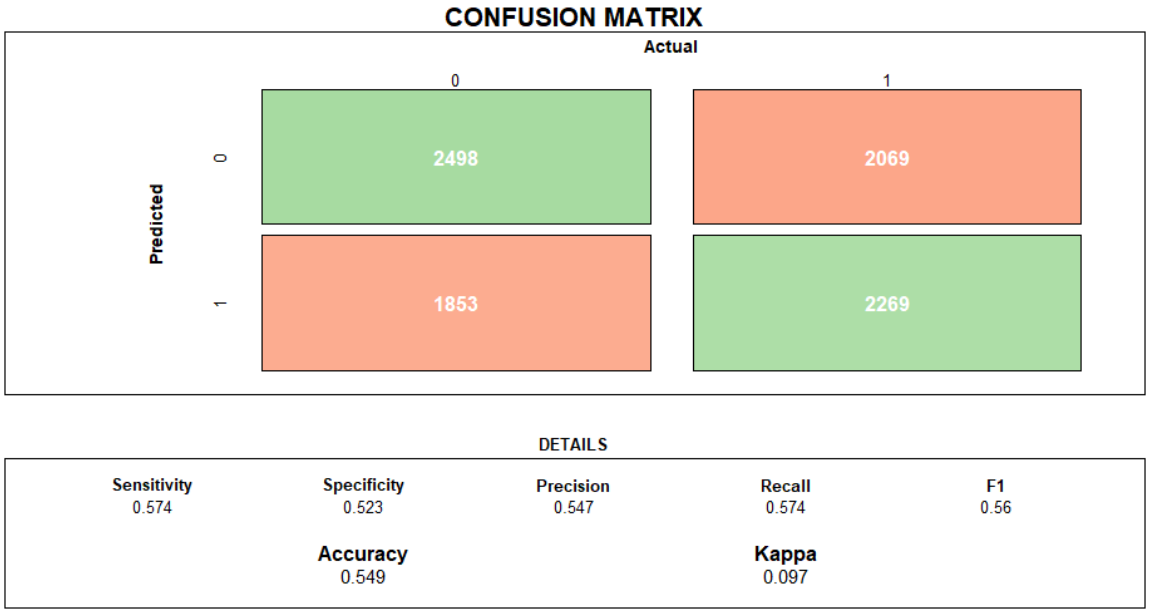

Figura 5.3: Matriz de confusión. Naive-Bayes pre-game.

Una vez más, la diferencia es clara cuando comparamos los resultados obtenidos con el modelo entrenado con el dataset al completo. Con los parámetros fL= 0, adjust = 1, usekernel = False, se obtiene una precisión = 0.8942 y RMSE = 0.32022

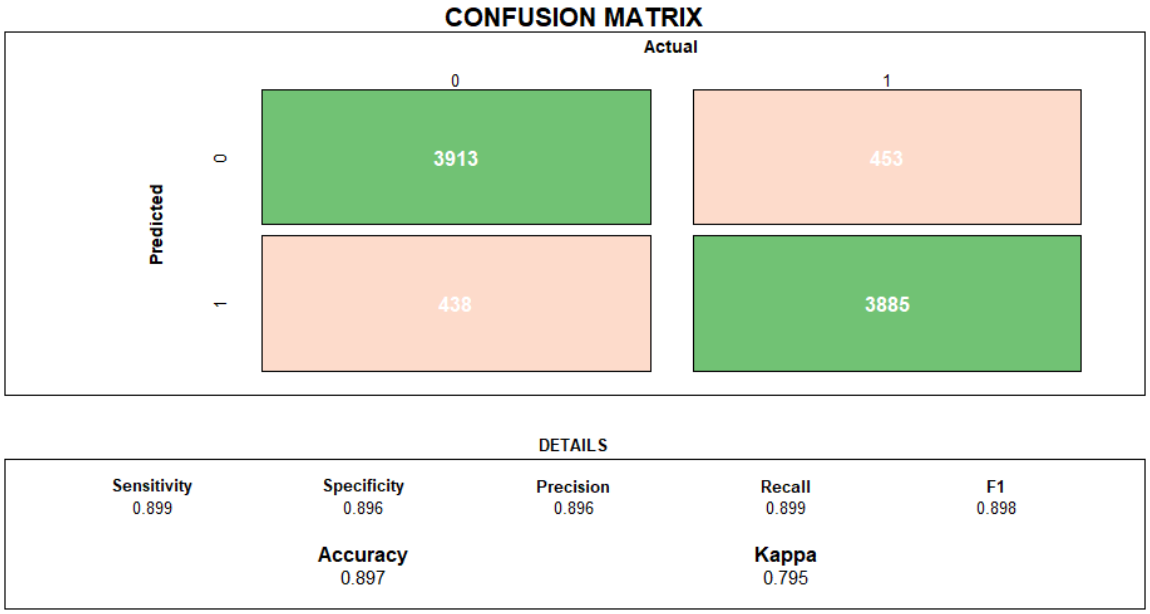

Figura 5.4: Matriz de confusión. Naive-Bayes completo.

5.1.3 KNN

Una de las variaciones más conocidas del algoritmo de vecino más próximo (NN) es el KNN. Consiste en tomar las k observaciones más parecidas a una categoría concreta. Los vecinos seleccionados son observaciones que han sido clasificadas previamente. KNN es un algoritmo de aprendizaje supervisado, una vez ha sido entrenado con un conjunto de datos, es capaz de clasificar correctamente instancias nuevas. KNN es un algoritmo eficaz cuando el número posible de categorías es alto o cuando la muestra es heterogénea. No obstante, es un algoritmo con un alto costo computacional.

Una vez más, el modelo entrenado con datos previos a la partida no es capaz de conseguir una tasa de predicción por encima del azar, precisión = 0.5406, k = 5, Kappa = 0.08127 y RMSE = 0.6628

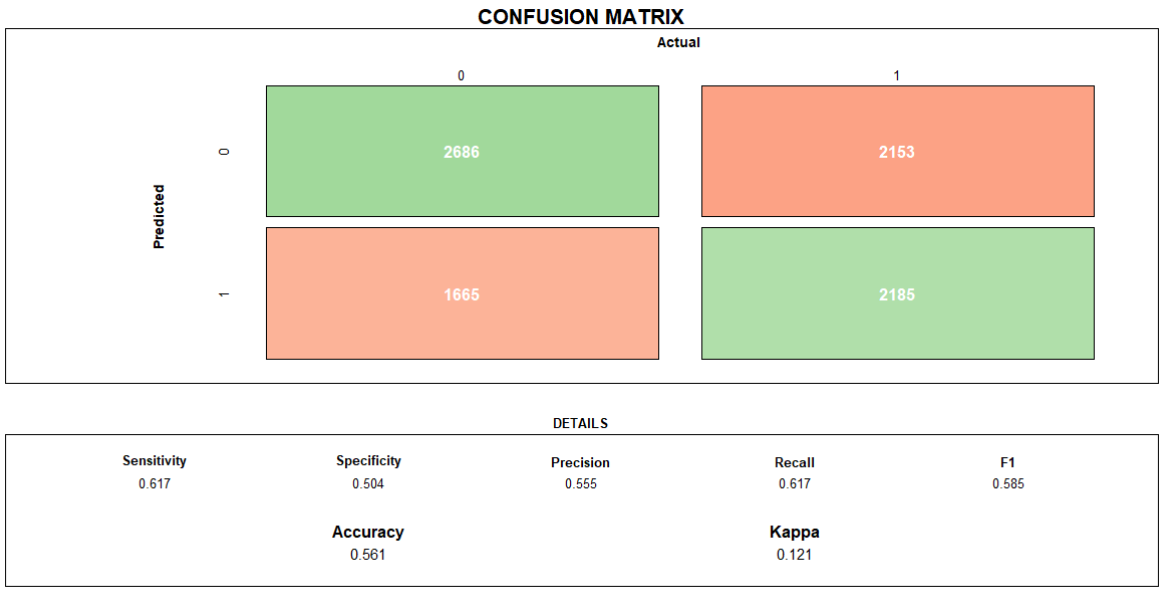

Figura 5.5: Matriz de confusión. knn pre-game.

No obstante, el modelo entrenado con el dataset al completo, devuelve con el parámetro k = 9, una precisión = 0.8963, Kappa = 0.7926 y RMSE = 0.31932

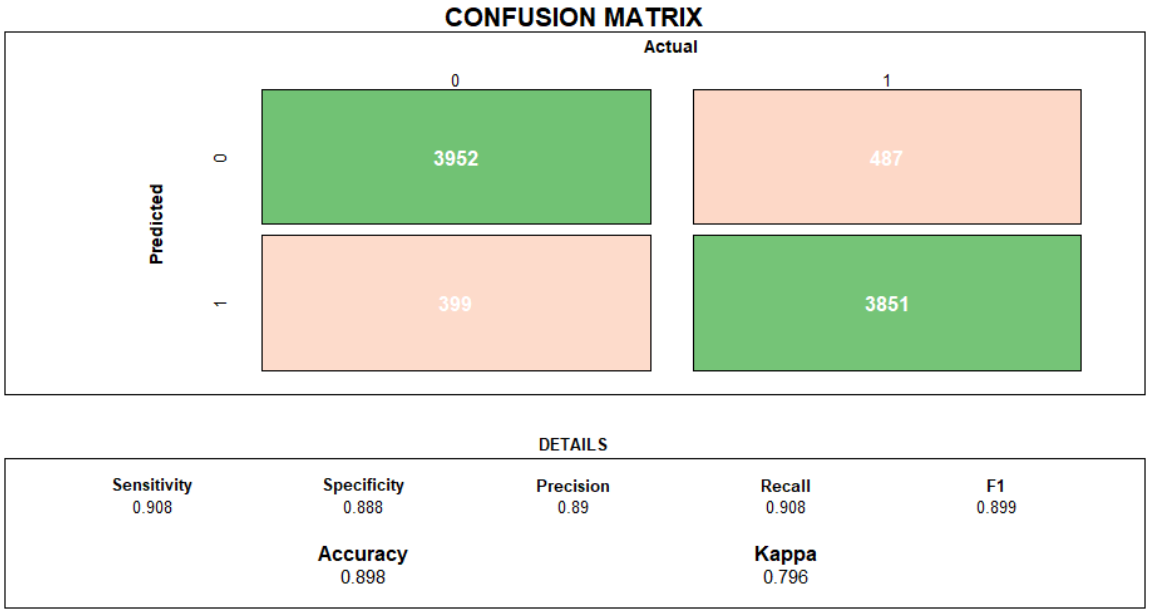

Figura 5.6: Matriz de confusión. Knn completo.

5.1.4 XGBoost

El "boosting" es un método que funciona de manera similar al "bagging". Se crean varias copias del dataset de entrenamiento asignando un árbol de decisión a cada copia y después combinándolas para obtener un único modelo predictivo. En el caso del "boosting" los árboles se construyen de manera secuencial: cada árbol se construye empleando información obtenida previamente de los árboles anteriores.

En concreto, el paquete XgBoost surge en 2016. La principal ventaja que presenta con respecto a iteraciones previas del "Gradient Boosting" es la regularización. Una técnica para prevenir el sobreajuste. También implementa procesamiento paralelo por lo que el tiempo de cálculo es menor y es capaz de procesar de manera interna valores missing.

La parametrización del XGBoost es el resultado de una rejilla que para ambos modelos presentados concluye en los siguientes parámetros: nfold = 10, max depth = 13, eta = 0.2, nthread 12, subsample = 0.5 y colsample bytree = 0.9

El modelo entrenado con los datos previos a la partida es el más prometedor de los que se presentan en este trabajo. Logra una precisión = 0.675 y RMSE = 0.570

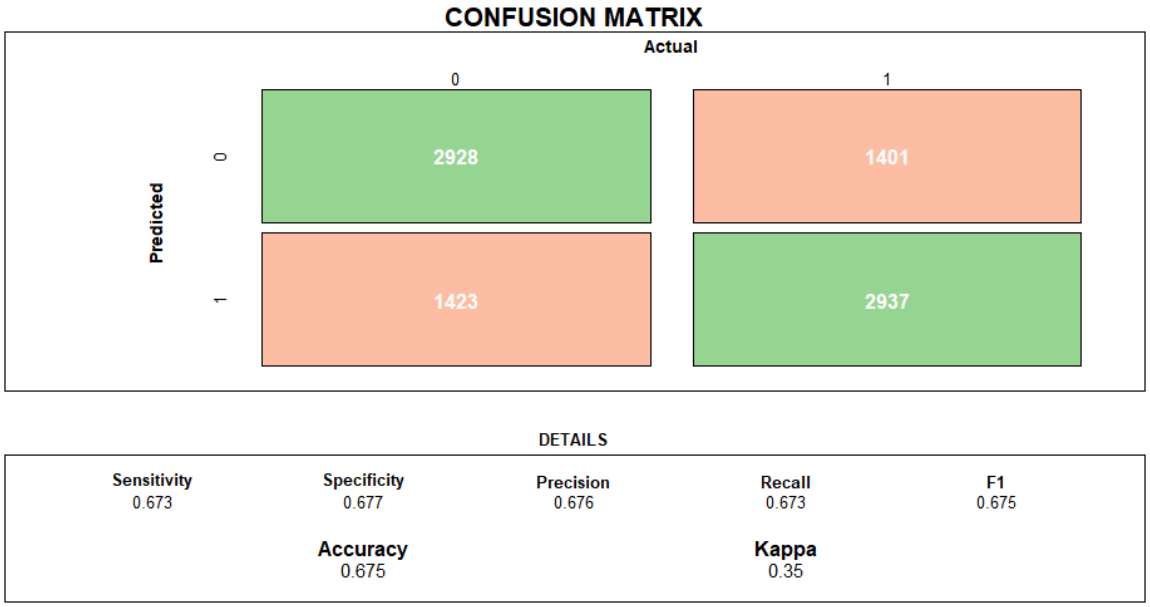

Figura 5.7: Matriz de confusión. XGBoost pre-game.

Por último, el modelo entrenado con los datos al completo, logra una precisión = 0.928 y RMSE = 0.267. Siendo el mejor modelo de la comparativa.

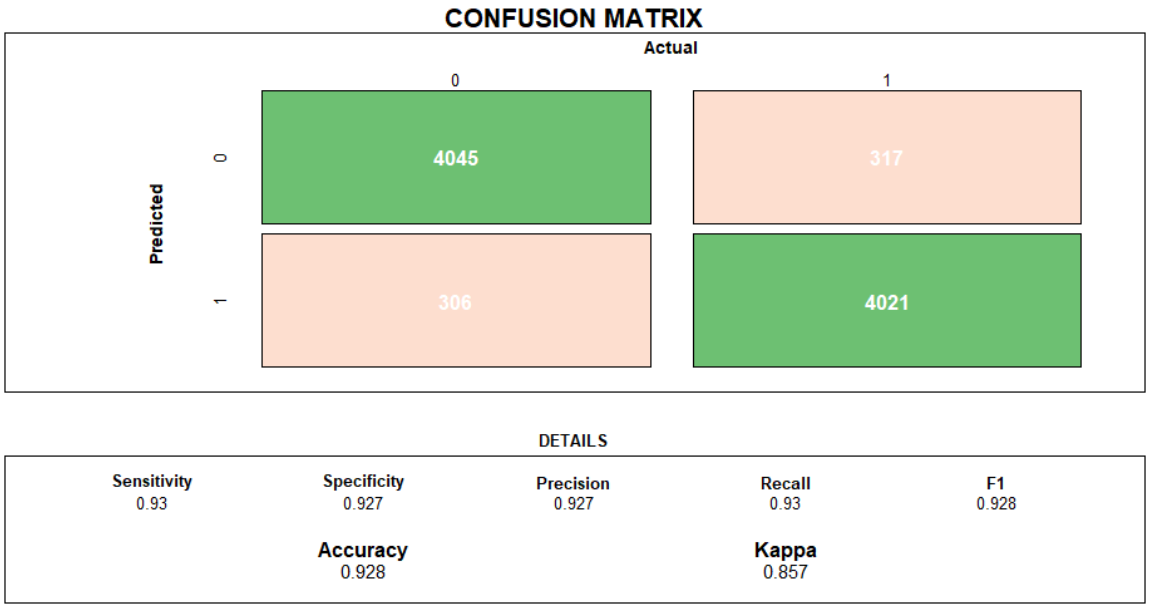

Figura 5.8: Matriz de confusión. XGBoost completo.

5.1.5 Random Forest

Random Forest supone una mejora sobre los métodos de "bagging" generando árboles decorrelados entre sí. Los árboles de decisión se construyen tomando para cada división una m muestra aleatoria de p predictores. Cada división utiliza uno de esos m predictores. Cada división toma una nueva muestra de esos m predictores. La cantidad de predictores escogida suele responder a $m \approx \sqrt{p}$.

Esta manera de limitar la cantidad de predictores que puede tomar el algoritmo cumple una función específica: si existe un predictor fuerte en el dataset, en los modelos de "bagging", la mayoría, sino todos los árboles, tendrán el predictor como medida principal, de forma que a la hora de comparar predicciones estarán todas correladas y no se da la reducción de varianza. Con este método, al eliminar los predictores fuertes de algunos de los árboles, se logra un error menor en OOB y en el test error.

Dado el coste computacional que tiene y la cantidad de modelos que se han tenido que entrenar, se ha implementado h2o [9]. h2o es una máquina virtual de Java optimizada para realizar procesos de machine learning en paralelo.

En la linea de lo ya expuesto, los modelos entrenados con el dataset previo al comienzo de la partida no logran una precisión mayor a 0.5339, profundidad media = 19.20, ntrees = 500 y RMSE = 0.49883

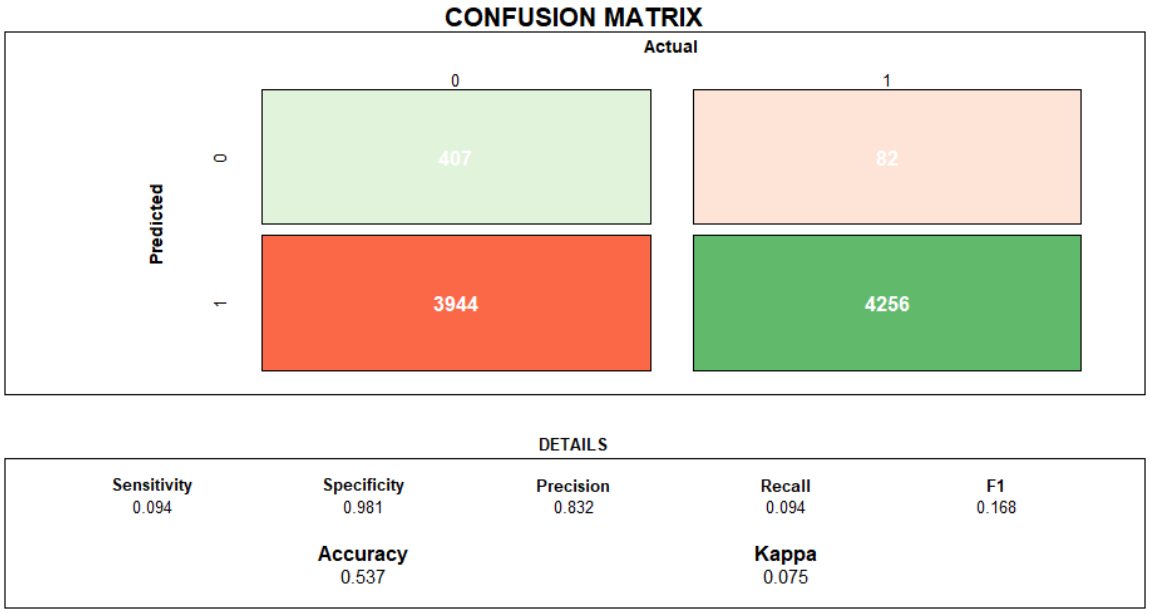

Figura 5.9: Matriz de confusión. Random Forest pre-game.

La precisión de los modelos aumenta drásticamente cuando los entrenamos con el dataset al completo. Obteniendo con los parámetros profundidad media = 19.2, ntrees = 500, una precisión = 0.896 y RMSE = 0.454

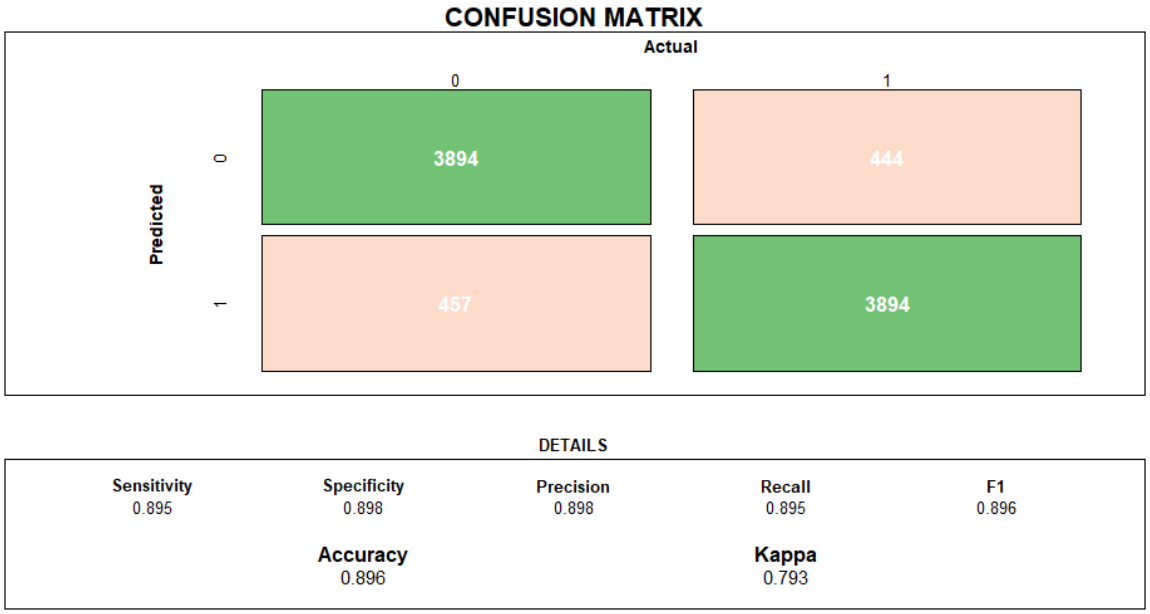

Figura 5.10: Matriz de confusión. Random Forest completo.

6 Discusión

Los objetivos de este proyecto eran: construir una base de datos suficientemente grande para entrenar modelos de clasificación en League of Legends, generar modelos eficientes con datos obtenidos a través de la API y tratar de conseguir estos modelos con los datos que se pudieran obtener previo al comienzo de la partida.

Los dos primeros objetivos se han cumplido: todos los algoritmos que se presentan en este trabajo son capaces de generar modelos de clasificación lo suficientemente robustos con los datos completos obtenidos de la API.

No obstante, los datos obtenidos previo al comienzo de la partida no han sido suficientes como para generar predicciones fiables.

Algoritmo	Pregame		Completo	
	RMSE	ACC	RMSE	ACC
$\mathbf{avNNet}_1$	0,659	0,565	0,305	0,900
$\mathbf{Nave-Bayes}_2$	0,671	0,549	0,320	0,897
$\mathbf{KNN}_3$	0,662	0,541	0,319	0,898
$\mathbf{XGBoost}_4$	0,570	0,675	0,267	0,928
$\mathbf{RandomForest}_5$	0,498	0,523	0,434	0,910

Cuadro 6.1: RMSE y precisión de los Algoritmos.

Existen ejemplos en la literatura capaces de generar modelos calculando la tasa de victorias que tiene cada jugador asociado al campeón que escoge para la partida [10]. Dadas las condiciones de uso que tiene el autor de este documento de la API de Riot Games, calcular esta medida predictiva hubiera sido demasiado costoso a nivel computacional y de acceso; por lo que en este proyecto intentamos generar una medida predictiva parecida a través de la maestría que tiene un jugador con un campeón, que supone un nivel de experiencia acumulado. En versiones previas y reducidas del dataset se lograron con XGBoost tasas de acierto en torno al 73 %.

Los modelos entrenados con los datasets completos resultan más potentes dado que incluyen información del proceso de la partida. Esto era algo esperado dado que es natural que los hitos logrados en la partida y registrados en el dataset influyan en la victoria/derrota. De estos modelos, XGBoost ha demostrado ser el más eficaz por su

velocidad de procesado y tasa de aciertos.

6.1 Conclusión

En este proyecto se ha presentado la metodología y el código para la construcción de una base de datos para generar modelos de predicción en League of Legends. Se logran modelos de clasificación eficientes con datos in-game, pero no con los datos que se obtienen previo al desarrollo de la misma.

6.2 Lineas futuras

En el futuro se podría mejorar el nivel de acceso a la API para replicar resultados con datos pre-game que generen mejores modelos predictivos, así como poner a prueba los modelos con otros conjuntos de datos pertenecientes a otras ligas del juego.

References

[1] Cómo jugar - league of legends.

[2] D. J. Hand, H. Mannila, and P. Smyth, *Principles of data mining (adaptive computation and machine learning)*. MIT Press, 2001.

[3] L. Lin, "League of legends match outcome prediction."

[4] A. Semenov, P. Romov, K. Neklyudov, D. Yashkov, and D. Kireev, "Applications of machine learning in dota 2: Literature review and practical knowledge sharing." in *MLSA@ PKDD/ECML*, 2016.

[5] R. T. d. Souza, "Aplicaçao de algoritmos classificadores para previsao de vitória em uma partida de league of legends," 2017.

[6] Kfc: Colonel ki.

[7] Cassiopeia documentation — cassiopeia 3.0.x documentation.

[8] avnnet function | r documentation.

[9] Overview — h2o 3.30.1.2 documentation.

[10] Lol predictor.

Anexo I: Código en Markdown

A continuación se adjunta la totalidad del código en R y Python necesario para realizar este proyecto en Markdown

Construcción de la base de datos en Python

La base de datos que se ha utilizado para este proyecto proviene de la API pública de Riot Games. Para realizar consultas a través de la API se ha utilizado el framework "Cassiopeia". Cassiopeia extiende la funcionalidad de un wrapper tradicional para facilitar el trabajo de consulta de manera intuitiva organizando las peticiones por clases.

Para obtener acceso a la API de Riot Games se requiere de una cuenta de desarrollador a la que se vincula una clave de desarrollo que dura 24h activa. Existen maneras de ampliar el acceso así como la tasa de consultas permitidas a sus servidores. Para ello habría que registrar el producto. Dado que la finalidad de este proyecto es investigación personal, estamos limitados a una clave personal. También se ha de fijar una región en la que trabajar, se escoge la región "Europa-West". Estos son requisitos básicos de la API para trabajar con ella

In [ ]:

```
cass.set_riot_api_key("clave-API")  # This overrides the value set in your configuration/settings.
cass.set_default_region("EUW")
```

Para acceder a los datos de una partida, se requiere del id de una partida. Esto se puede obtener de varias maneras: en la documentación de la API de Riot Games se pueden encontrar listas de ids para pruebas o se puede acceder al historial de partidas de un jugador.

El objetivo de esta parte del proyecto es conseguir la suficiente cantidad de ids de partidas de una liga específica para analizar los datos posteriormente en R. Para ello, podemos consultar en la API la lista de todos jugadores que pertenecen a la liga objetivo: másters. NO solo eso, también podemos especificar la modalidad de partida de interés, que en nuestro caso son partidas clasificatorias o "ranked".

In [ ]:

```
masters = cass.get_master_league(queue=Queue.ranked_solo_fives, region="EUW")
```

El output es la lista de identificadores de jugadores que pertenecen a esa liga en concreto. A partir de esta lista podemos consultar de cada uno de ellos el historial de partidas. Cada una de esas partidas lleva un id asociado que es el que usaremos para construir la base de datos. El output de la solicitud es un JSON sobre el que se ha de iterar. Para ello se definen dos bucles for que rellenan las listas "jugadores_master" e "historias". En la lista historias se alamacenan los ids de partida

In [ ]:

```
jugadores_master=list()
historias=list()

contador = 1
for master in masters:
    print("{parcial}/{total} - {summoner}".format(parcial=contador, total=len(masters), summoner=ma
ster.summoner.name))
    jugadores_master.append(master.summoner.name)

    summoner      = cass.get_summoner(name = master.summoner.name, region="EUW")
    match_history = summoner.match_history(seasons={Season.season_9},
queues={Queue.ranked_solo_fives})
    #match_history(seasons={Season.season_9}, queues={Queue.ranked_solo_fives})


    for historia in match_history:
        print(historia)
        historias.append(historia.id)

    contador = contador + 1
```

Una vez tenemos la lista de ids, hablaremos del proceso de consulta de una partida. Cassiopeia tiene la función "cass.get_match()" cuyos argumentos son el id de la partida y la región en la que se consulta. El output de la función es un JSON que contiene mucha información de la que necesitamos apartados específicos y estructurados. Para automatizar este proceso se han escrito una serie de funciones anidadas es una función grande que procesará las partidas de una a una y almacenará el resultado en un formato csv listo para exportar a R.

La función grande se ha denominado "procesar_partida". La función requiere de un solo argumento:

La función grande se ha denominado "procesar_partida". La función requiere de un solo argumento:

- id_partida: el id de la partida.

La función almacena en listas los datos del JSON que nos son de interés y luego los almacena respetando la estructura en una lista separada por comas como un csv. El resultado son 42 columnas por cada id de partida.

In [ ]:

```
def procesar_partida(id_partida):
    blue_wins          = list() #Lista principal
    first_baron        = list()
    first_dragon       = list()
    first_blood        = list()
    first_inhibitor    = list()
    first_tower        = list()
    first_rifth        = list()
    tower_kills        = list()
    dragon_kills       = list()
    baron_kills        = list()
    inhibitor_kills    = list()
    rifth_kills        = list()
    picks_lista        = list()
    bans_lista         = list()
    maestrias_lista    = list()
    partida            = cass.get_match(id_partida, region = "EUW")
    equipos            = partida.teams
    participantes      = partida.participants

    # TODO: GetOro(participantes, oro_lista)

    procesar_booleano(partida.blue_team.win                    , blue_wins        )
    procesar_booleano(partida.blue_team.first_baron            , first_baron      )
    procesar_booleano(partida.blue_team.first_dragon           , first_dragon     )
    procesar_booleano(partida.blue_team.first_blood            , first_blood      )
    procesar_booleano(partida.blue_team.first_inhibitor        , first_inhibitor  )
    procesar_booleano(partida.blue_team.first_tower            , first_tower      )
    procesar_booleano(partida.blue_team.first_rift_herald      , first_rifth      )

    procesar_numerico(partida.blue_team.tower_kills            , tower_kills      )
    procesar_numerico(partida.blue_team.dragon_kills           , dragon_kills     )
    procesar_numerico(partida.blue_team.baron_kills            , baron_kills      )
    procesar_numerico(partida.blue_team.inhibitor_kills        , inhibitor_kills  )
    procesar_numerico(partida.blue_team.rift_herald_kills      , rifth_kills      )

    for team in equipos:
        for ban in team.bans:
            if ban is None:
                bans_lista.append('0')
            else:
                bans_lista.append(ban.id)
    for participante in participantes:
        picks_lista.append(participante.champion.id)
        maestria   = maestria_campeon(participante,participante.champion.name)
        maestrias_lista.append(maestria)
    partida_procesada = blue_wins + first_dragon + first_baron + first_blood + first_inhibitor + fi
rst_tower + first_rifth + tower_kills + dragon_kills + baron_kills + inhibitor_kills + rifth_kills
+ picks_lista +bans_lista +maestrias_lista
    return(partida_procesada)
```

La función lleva anidadas tres funciones escritas para automatizar este proceso además de la función propia de Cassiopeia "cass.get_match()". Son las funciones: maestria_campeon, procesar_booleano y procesar_numerico.

La función maestria_campeon es una consulta separada a la API. Cada jugador tiene un nivel de maestria asociado a cada campeon del juego. La función coge el identificador del campeon que ha escogido el jugador y el id del jugador para consultar el nivel de maestria. La función tiene dos argumentos:

- participante: el id del jugador que escoge el campeón.
- champion_name: el nombre del campeon escogido

La función realiza la consulta y devuelve un número: el nivel de maestría.

In [ ]:

```
def maestria_campeon(participante, champion_name):
```

```
                contador = contador + 1
            file.write("\n")
        file.close()
```

Avnnet

Diego Candela Salomón

13/9/2020

#Avnnet

En este documento mostramos la implementación y entrenamiento de dos modelos entrenados con caret de avnnet.

Los paquetes que se utilizaron son los siguientes:

```
library(caret)
library(dplyr)
```

Para esta parte del análisis se ha diseñado una función muy simple. Dado que los parámetros de entrenamiento se han encontrado a través de ensayo y error, se requería una función con la que averiguar rápidamente la eficacia del modelo previo a construir una matriz de confusión.

La función aciertos_avnnet requiere de un argumento:

- df: un dataframe que contiene en la primera columna las etiquetas objetivo y en la segunda columna las predicciones resultantes del modelo.

La función devuelve un dataframe con una columna codificada en Verdadero/Falso para averiguar rápidamente los aciertos del modelo.

```
aciertos_avnnet <- function(df){
  df<- as.data.frame(df)
  df$Aciertos <- F
  for (i in 1:length(df$Aciertos)){
    if ((df[i,1] == df[i,2])){
      df[i,3] <- T
    }
  }
  return(df)
}
```

El paquete caret permite entrenar modelos de avnnet que son modelos de redes neuronales promediadas. Primero importaremos el dataset previo a la partida y después el dataset completo. Entrenaremos dos modelos.

Primero realizamos la importación de los datasets data_train, data_test y sus correspondientes etiquetas. Hacemos un cbind del data_train con sus etiquetas.

```
data_train         <- readRDS("./datosPreGame/data_train")
data_test          <- readRDS("./datosPreGame/data_test")
labels_data_train  <- readRDS("./datosPreGame/labels_train")
labels_data_test   <- readRDS("./datosPreGame/labels_test")

data_train <- cbind(labels_data_train, data_train)
```

Los modelos de avnnet suelen tener problemas cuando trabajan con muchos predictores binarios, especialmente cuando la varianza de muchos de estos predictores es próxima a cero. Con este motivo, el paquete caret incluye una función "nearZeroVar" que genera un índice de predictores que cumplen estas características y que han de ser eliminados de dataset para que el modelo se calcule correctamente.

```
indiceCerosTrain <- nearZeroVar(data_train)

data_train       <- data_train[,-indiceCerosTrain]
```

##Preparando caret

Una vez el dataset está listo procedemos a establecer los parámetros de caret. A lo largo de los distintos modelos que se han generado en las pruebas de este trabajo, estos son los parámetros escogidos.

```
grupos=4
sinicio=1234
repe=3
size=c(5)
decay=c(0.01)
repeticiones=3
itera=100
```

Y ahora establecemos los parámetros de control, el grid y el entrenamiento del modelo.

```
control<-trainControl(method = "repeatedcv",number=grupos,repeats=repe,
                      savePredictions = "all",classProbs=TRUE)

# Aplico caret y construyo modelo

avnnetgrid <-  expand.grid(size=size,decay=decay,bag=FALSE)

avnnet_1<- train(make.names(labels_data_train)~.,
                 data=data_train,
                 method="avNNet",
                 linout = FALSE,
                 maxit=itera,
                 repeats=repeticiones,
                 trControl=control,
                 tuneGrid=avnnetgrid,
                 trace=T)
```

Una vez se ha entrenado el modelo se ha de poner a prueba frente al data_test. Los resultados están expuestos en el documento principal de este proyecto.

Primero aplicamos el mismo índice de la función "nearZeroVar" al set de testeo y luego utilizamos la función del paquete base de R "predict".

Guardamos esas predicciones en un dataframe para generar la matriz de confusión con la función proporcionada por el paquete caret.

Con el fin de asegurar que no ha habido ningún cambio de formato y que el vector predictions tiene los mismos niveles que las etiquetas del data_test añadimos la linea 110 y generamos la matriz de confusión.

```
data_test <- cbind(labels_data_test, data_test)
data_test <- data_test[,-indiceCerosTrain]
data_test <- data_test[,-1]

predictions  <- predict(avnnet_1, data_test)
predicciones <- as.data.frame(cbind(labels_data_test, predictions))

levels(predictions) <- c("0","1")

matriz_confusion<- confusionMatrix(data=predictions, reference = labels_data_test)
```

Una vez obtenemos la matriz de confusión, averiguamos el RMSE para valorar el modelo en su conjunto en la comparación con el resto de modelos incluidos en este trabajo

```
residuals_preGame <- as.numeric(labels_data_test) - as.numeric(predictions)
RMSE_preGame      <- sqrt(mean(residuals_preGame^2))
```

Una vez queda calculado el modelo, eliminamos del entorno de trabajo tanto data_train y data_test para importar las versiones completas

```
rm(data_train,data_test)
```

Importamos las versiones completas de los respectivos datasets

```
data_train         <- readRDS("./datosCompleto/data_train")
data_test          <- readRDS("./datosCompleto/data_test")

data_train       <- cbind(labels_data_train, data_train)
indiceCerosTrain <- nearZeroVar(data_train)
data_train       <- data_train[,-indiceCerosTrain]
```

Y volvemos a entrenar el modelo respetando los mismos parámetros para la comparación final

```
control<-trainControl(method = "repeatedcv",number=grupos,repeats=repe,
                      savePredictions = "all",classProbs=TRUE)

avnnetgrid <-  expand.grid(size=size,decay=decay,bag=FALSE)

avnnet_2<- train(make.names(labels_data_train)~.,
                 data=data_train,
                 method="avNNet",
                 linout = FALSE,
                 maxit=itera,
                 repeats=repeticiones,
                 trControl=control,
                 tuneGrid=avnnetgrid,
                 trace=T)
```

Volvemos a realizar el mismo proceso para extraer las predicciones, la matriz de confusión y el RMSE

```
data_test <- cbind(labels_data_test, data_test)
data_test <- data_test[,-indiceCerosTrain]
data_test <- data_test[,-1]

predictions  <- predict(avnnet_2, data_test)
predicciones <- as.data.frame(cbind(labels_data_test, predictions))

levels(predictions) <- c("0","1")

matriz_confusion<- confusionMatrix(data=predictions, reference = labels_data_test)

residuals_completo <- as.numeric(labels_data_test) - as.numeric(predictions)
RMSE_completo      <- sqrt(mean(residuals_completo^2))
```

K-Nearest Neighbors

Diego Candela Salomón

14/9/2020

##K-nearest Neighbors

Knn es uno de los algoritmos más empleados en problemas de clasificación. Se puede utilizar caret para entrenar modelos de clasificación en knn.

###Funciones

Dado que las predicciones de un modelo knn se devuelven como dos columnas con la probabilidad calculadada de pertenecer a cada categoría, se escribe una función para igualar el formato de las predicciones con el de las etiquetas del data_test.

La función aciertos_a_confusión tiene un argumento:

- df: el dataframe de predicciones.

El ouput de la función es el mismo dataframe con una columna con las predicciones del modelo igualadas al formato de las etiquetas.

```
aciertos_a_confusion <- function(df){
  df <- as.data.frame(df)
  df$aciertos <- 0
  for (i in 1:length(df$aciertos)){
    if ((df[i,3] > df[i,2])){
      df[i,4] <- 1
    }
  }
  return(df)
}
```

###Importar datos

Dado que el formato de los datos se mantiene para el resto de algoritmos de ahora en adelante, el proceso de importación es el mismo que hemos visto hasta ahora. Generaremos dos modelos a partir de sus correspondientes conjuntos de datos. Tras importar el conjunto de datos recogidos antes del comienzo de la partida, realizamos un cbind de las etiquetas de data_train con data_train

```
data_train         <- readRDS("./datosPreGame/data_train")
data_test          <- readRDS("./datosPreGame/data_test")
labels_data_train  <- readRDS("./datosPreGame/labels_train")
labels_data_test   <- readRDS("./datosPreGame/labels_test")

data_train <- cbind(labels_data_train, data_train)
```

Dada la cantidad de predictores que tiene el dataset y teniendo en cuenta que se encuentran en formato dummy, knn tiene problemas para generar un modelo cuando las variables tienen varianza próxima a cero. Para esto caret dispone de la función "nearZeroVar" que genera un índice de variables que cumplen estas caracterísitas para poder eliminarlas del dataset

```
indiceCerosTrain <- nearZeroVar(data_train)

data_train       <- data_train[,-indiceCerosTrain]
```

###Preparando caret

Una vez los datos están preparados para entrenar el modelo, preparamos los parámetros de caret para knn

```
control <- trainControl(method = "repeatedcv",
                        repeats = 3,
                        verboseIter = T)

knn_1 <- train(labels_data_train~.,
               data_train,
               'knn',
               trControl = control,
               tuneLength = 20
)
```

Una vez se ha entrenado el modelo se ha de poner a prueba frente al data_test. Los resultados están expuestos en el documento principal de este proyecto.

Primero aplicamos el mismo índice de la función "nearZeroVar" al set de testeo y luego utilizamos la función del paquete base de R "predict". La función ha de ser ajustada para el parámetro k óptimo obtenido en el modelo

Guardamos esas predicciones en un dataframe para generar la matriz de confusión con la función proporcionada por el paquete caret una vez pasamos esas predicciones por la función aciertos_a_confusion.

Con el fin de asegurar que no ha habido ningún cambio de formato y que el vector predictions tiene los mismos niveles que las etiquetas del data_test añadimos la linea 110 y generamos la matriz de confusión.

```
data_test <- cbind(labels_data_test, data_test)
data_test <- data_test[,-indiceCerosTrain]
data_test <- data_test[,-1]

levels(labels_data_test) <- <- c("0","1")

predictions  <- predict(knn_1,
                        newdata = data_test,
                        k = 9,
                        type = "prob",
                        prob=T)

predicciones <- as.data.frame(cbind(labels_data_test, predictions))

probs1 <- aciertos_a_confusion(predicciones)

matriz_confusion <- confusionMatrix(data      = as.factor(probs1[,4]),
                                    reference = as.factor(probs1[,1]))
```

Una vez hemos generado la matriz de confusión podemo proceder a realizar el cálculo del RMSE para la comparación de los modelos.

```
residuals_preGame <- as.numeric(labels_data_test) - as.numeric(probs1[,4])
RMSE_preGame      <- sqrt(mean(residuals_preGame^2))
```

Una vez tenemos el primer modelo, podemos proceder a eliminar los datasets del entorno de trabajo para importar los datasets completos y entrenar el segundo modelo

```
rm(data_train,data_test)

data_train         <- readRDS("./datosCompleto/data_train")
data_test          <- readRDS("./datosCompleto/data_test")

data_train <- cbind(labels_data_train, data_train)
```

Repetimos el mismo proceso para generar el índice de variables con varianza cercanas a 0

```
indiceCerosTrain <- nearZeroVar(data_train)

data_train       <- data_train[,-indiceCerosTrain]
```

Y podemos proceder a generar el segundo modelo con los mismos parámetros

```
control <- trainControl(method = "repeatedcv",
                        repeats = 3,
                        verboseIter = T)

knn_2 <- train(labels_data_train~.,
               data_train,
               'knn',
               trControl = control,
               tuneLength = 20
)
```

Realizamos el mismo proceso para las predicciones, matriz de confusión y cálculo del RMSE

```
predictions  <- predict(knn_2,
                        newdata = data_test,
                        k = 9,
                        type = "prob",
                        prob=T)

predicciones <- cbind(labels_data_test, predictions)

predicciones <- as.data.frame(cbind(labels_data_test, predictions))

probs1 <- aciertos_a_confusion(predicciones)

matriz_confusion <- confusionMatrix(data      = as.factor(probs1[,4]),
                                    reference = as.factor(probs1[,1]))

residuals_completo <- as.numeric(labels_data_test) - as.numeric(probs1[,4])
RMSE_completo      <- sqrt(mean(residuals_completo^2))
```

```
matriz_confusion <- confusionMatrix(data      = as.factor(probs1[,4]),
                                    reference = as.factor(probs1[,1]))

residuals_completo <- as.numeric(labels_data_test) - as.numeric(probs1[,4])
RMSE_completo      <- sqrt(mean(residuals_completo^2))
```

Naïve-Bayes

Diego Candela Salomón

14/9/2020

##Naïve-Bayes

Naïve-Bayes es uno de los algoritmos más sugeridos en la literatura para problemas de clasificación. El paquete caret dispone de una función para entrenar modelos de clasificación con este algoritmo.

###Importar datos

Dado que el formato de los datos se mantiene para los algoritmos, el proceso de importación es el mismo que hemos visto hasta ahora. Generaremos dos modelos a partir de sus correspondientes conjuntos de datos. Tras importar el conjunto de datos recogidos antes del comienzo de la partida, realizamos un cbind de las etiquetas de data_train con data_train

```
data_train         <- readRDS("./datosPreGame/data_train")
data_test          <- readRDS("./datosPreGame/data_test")
labels_data_train  <- readRDS("./datosPreGame/labels_train")
labels_data_test   <- readRDS("./datosPreGame/labels_test")

data_train <- cbind(labels_data_train, data_train)
```

Dada la cantidad de predictores que tiene el dataset y teniendo en cuenta que se encuentran en formato dummy, knn tiene problemas para generar un modelo cuando las variables tienen varianza próxima a cero. Para esto caret dispone de la función "nearZeroVar" que genera un índice de variables que cumplen estas característas para poder eliminarlas del dataset

```
indiceCerosTrain <- nearZeroVar(data_train)

data_train       <- data_train[,-indiceCerosTrain]
```

###Preparando caret

Una vez los datos están preparados para entrenar el modelo, preparamos los parámetros de caret para nb

```
#NAIVE BAYES
model_1 <- train(labels_data_train~.,
                 data_train,
                 'nb',
                 trControl = trainControl(method = 'cv', number=10))
```

Una vez se ha entrenado el modelo se ha de poner a prueba frente al data_test. Los resultados están expuestos en el documento principal de este proyecto.

Primero aplicamos el mismo índice de la función "nearZeroVar" al set de testeo y luego utilizamos la función del paquete base de R "predict".

Guardamos esas predicciones en un dataframe para generar la matriz de confusión con la función proporcionada por el paquete caret.

```
data_test <- cbind(labels_data_test, data_test)
data_test <- data_test[,-indiceCerosTrain]
data_test <- data_test[,-1]

predictions  <- predict(model_1, data_test)
predicciones <- as.data.frame(cbind(labels_data_test, predictions))

confusion_matriz<- confusionMatrix(data=predictions, reference = labels_data_test)
```

Una vez hemos generado la matriz de confusión podemo proceder a realizar el cálculo del RMSE para la comparación de los modelos.

```
residuals_preGame <- as.numeric(labels_data_test) - as.numeric(predictions)
RMSE_preGame      <- sqrt(mean(residuals_preGame^2))
```

Una vez tenemos el primer modelo, podemos proceder a eliminar los datasets del entorno de trabajo para importar los datasets completos y entrenar el segundo modelo

```
rm(data_train,data_test)

data_train         <- readRDS("./datosCompleto/data_train")
data_test          <- readRDS("./datosCompleto/data_test")

data_train <- cbind(labels_data_train, data_train)
```

Repetimos el mismo proceso para generar el índice de variables con varianza cercanas a 0

```
indiceCerosTrain <- nearZeroVar(data_train)

data_train       <- data_train[,-indiceCerosTrain]
```

Y podemos proceder a generar el segundo modelo con los mismos parámetros

```
model_2 <- train(labels_data_train~.,
                 data_train,
                 'nb',
                 trControl = trainControl(method = 'cv', number=10))
```

Realizamos el mismo proceso para las predicciones, matriz de confusión y cálculo del RMSE

```
predictions  <- predict(knn_2,
                        newdata = data_test,
                        k = 9,
                        type = "prob",
                        prob=T)

predicciones <- cbind(labels_data_test, predictions)

predicciones <- as.data.frame(cbind(labels_data_test, predictions))

probs1 <- aciertos_a_confusion(predicciones)
```

Random Forest

Diego Candela Salomón

15/9/2020

Random Forest en h2o

Random Forest es de los algoritmos más frecuentes en las competiciones de datos. Es un algoritmo agresivo capaz de detectar interacciones entre grandes conjuntos de variables.

Debido al alto coste computacional que tiene Random Forest, se ha implementado h2o. H2o es una máquina virtual de Java especialiada en clusters de procesamiento paralelo para operaciones matemáticas

Los paquetes que se requieren para el cálculo son

```
library(h2o)
library(randomForest)
library(caret)
library(bit64)
```

El grueso del proceso ocurre en los servidores de h2o una vez ya se ha configurado todo. Así pues importamos los datos con el mismo método que para el resto de algoritmos. Primero importamos el dataset de los datos previos a la partida.

```
data_train         <- readRDS("./datosPreGame/data_train")
data_test          <- readRDS("./datosPreGame/data_test")
labels_data_train  <- readRDS("./datosPreGame/labels_train")
labels_data_test   <- readRDS("./datosPreGame/labels_test")

data_train <- cbind(labels_data_train, data_train)
```

Se ha de iniciar el cluster para conectar con h2o

```
h2o.init()
```

h2o requiere que los datos se codifiquen en su formato específico y no como un dataframe para realizar los cálculos. Esto será una constante durante el tiempo que utilicemos h2o.

Preparamos el modelo con los parámetros para el entrenamiento. Nótese que en el training_frame se ha de hacer la conversión a h2o:

```
rf_1 <-  h2o.randomForest(
  x = colnames(data_train)[2:ncol(data_train)],
  y = colnames(data_train)[1],
  training_frame = as.h2o(data_train, use_datatable = T),
  nfolds = 2,
```

```
  mtries = 4,
  ntrees = 500,
  verbose= T
)
```

Una vez obtenido el primer modelo eliminamos data_train y data_test para importar las versiones completas del dataset

```
data_train         <- readRDS("./datosCompleto/data_train")
data_test          <- readRDS("./datosCompleto/data_test")

data_train <- cbind(labels_data_train, data_train)
```

Volvemos a entrenar un modelo con los mismos parametros

```
rf_2 <-  h2o.randomForest(
  x = colnames(data_train)[2:ncol(data_train)],
  y = colnames(data_train)[1],
  training_frame = as.h2o(data_train, use_datatable = T),
  nfolds = 2,
  mtries = 4,
  ntrees = 500,
  verbose= T
)
```

Una vez tenemos entrenados los modelos, los almacenamos. El proceso de guardado difiere del método de captura de imágenes de RStudio y requiere de comandos específicos de h2o.

```
h2o.saveModel("./rf_1_preGame")
h2o.saveModel("./rf_1_completo")
```

Posteriormente cargaremos los modelos para realizar las predicciones. Para ello hemos de volver a convertir del formato h2o al dataframe de R.

```
modelo_cargado_1    <- h2o.loadModel(path = "./rf_1_Completo/DRF_model_R_1599602428198_249")
predictions_2       <- h2o.predict(object = modelo_cargado_1,
                           newdata = as.h2o(data_test, use_datatable = T))
predicciones_bien_1 <- as.data.frame(predictions_2)

rm(data_test)

##  CARGAMOS EL SEGUNDO MODELO PARA PREDECIR.
data_test           <- readRDS("./datosPreGame/data_test")
modelo_cargado_2    <- h2o.loadModel(path = "./rf_1_preGame/DRF_model_R_1599602428198_1")

predictions_1       <- h2o.predict(object = modelo_cargado_2,
                           newdata = as.h2o(data_test, use_datatable = T))
predicciones_bien_2 <- as.data.frame(predictions_1)
```

Una vez hemos generado los dataframes de predicciones podemos proceder a realizar las matrices de confusión

```
probs1 <- as.data.frame(cbind(as.factor(labels_data_test), as.factor(predicciones_bien_1[,1])))
probs2 <- as.data.frame(cbind(as.factor(labels_data_test), as.factor(predicciones_bien_2[,1])))

probs1[,2] <- as.factor(probs1[,2])
levels(probs1[,2]) <- c("0","1")

probs2[,2] <- as.factor(probs2[,2])
levels(probs2[,2]) <- c("0","1")


matriz_confusion_1 <- confusionMatrix(data      = as.factor(probs1[,2]),
                                      reference = labels_data_test)
matriz_confusion_2 <- confusionMatrix(data      = as.factor(probs2[,2]),
                                      reference = labels_data_test)
```

XGBoost

Diego Candela Salomón

15/9/2020

XGBoost y Rejilla

El paquete XGBoost ha cogido mucha popularidad como un algoritmo agresivo capaz de encontrar interacciones en grandes cantidades de datos muy heterogéneos. No usaremos el paquete de caret para el entrenamiento de los modelos por lo que la parametrización corre a cargo del uso de una rejilla.

Los paquetes necesarios para este código son:

```
library(xgboost)
library(caret) #Para generar la matriz de confusión.
```

Funciones

para el output de la rejilla se han escrito dos funciones muy simples que etiquetaran el csv resultante de las predicciones con la tasa de aciertos para escoger un modelo ganador.

La función "Aciertos" tiene un argumento:

- df: el dataframe de predicciones.

Dado que el output de la función "predict" del paquete base de R devuelve para un objeto XGBoost la probabilidad respectiva de que la clasificación sea 0 y 1, la función "Aciertos" compara las probabilidades y la iguala al formato de las etiquetas del data_test.

El output es el mismo dataframe con una columna codificada en 0 y 1 igual al formato de las etiquetas del data_test.

```
Aciertos <- function(df){
  df$Aciertos <- F
  for (i in 1:length(df$Aciertos)){
    if ((df[i,2] > df[i,3] && df[i,1] == 1) || (df[i,2] < df[i,3] && df[i,1] == 0)){
      df[i,4] <- T
    }
  }
  return(df)
}
```

La función "tasa" calcula la tasa de aciertos a partir del dataframe resultante de la función "Aciertos". Tiene un solo argumento:

- dfaciertos: el dataframe resultante de la función "Aciertos"

La función calcula la tasa de predicciones acertadas por el modelo de XGBoost. El output es un número flotante.

```
tasa <- function(dfaciertos){
  Trues  <- sum(dfaciertos$Aciertos == T)
  Falses <- sum(dfaciertos$Aciertos == F)
  result <- (Trues/(Trues+Falses))*100
  return(result)
}
```

Se escribe una última función para igualar el formato de las predicciones al que demanda la función "confusionMatrix" del paquete caret. La función "Aciertos_a_confusion" tiene un argumento:

- df: el dataframe de predicciones.

Dado que el output de la función "predict" del paquete base de R devuelve para un objeto XGBoost la probabilidad respectiva de que la clasificación sea 0 y 1, la función "Aciertos_a_confusion" iguala el formato al que exige la función "confusionMatrix" para generar posteriormente una matriz de confusión.

```
Aciertos_a_confusion <- function(df){
  df$Aciertos <- 0
  for (i in 1:length(df$Aciertos)){
    if ((df[i,2] > df[i,3] )){
      df[i,4] <- 1
    }
  }
  return(df)
}
```

Rejilla

Para encontrar los parámetros óptimos para entrenar un modelo XGBoost, se ha escrito un bucle for anidado de 4 niveles. Se importan los datos de entrenamiento y testeo y se inicia el bucle volcando los resultados de las predicciones en csv que se almacenan en un fichero

```
depth <- c(15, 13, 11, 9, 7, 5, 3)
eta <- c(20:2) / 100
subsample <- c(5:10) / 10
colsample <- c(5:10) / 10

for (d in depth){
  for (e in eta){
    for (s in subsample){
      for (c in colsample){

        bst_model <- xgboost(data = predictors,
                             nfold = 10,
                             early_stopping_rounds =  30,
                             label = as.matrix(target),
                             num_class = 2,
                             max_depth = d,
                             eta = e,
```

```
                            nthread = 12,
                            subsample = s,
                            colsample_bytree = c,
                            min_child_weight = 1,
                            nrounds = 600, # PARA EL TESTEO LO HAS BAJADO A 20. ESTABA A 600
                            objective = "multi:softprob",
                            maximize = FALSE)

        predictions  <- predict(bst_model, as.matrix(data_test))
        results <- data.frame(predictions[(1:length(predictions)) %% 2 == 0],
                              predictions[(1:length(predictions)) %% 2 == 1])

        resultscheck <- cbind(labels, results)
        aciertos     <- Aciertos(resultscheck)
        tasacheck    <- tasa(aciertos)

        #Aqui va la linea de write.csv con los parametros y con el df aciertos.
        write.csv(aciertos, file = paste0("./cross_validationXGBoost/results_depth_", d, "_eta_", e, "_s
      }
    }
  }
}
```

Modelos XGBoost

Una vez se conocen los parámetros de XGBoost para entrenar los modelos, se importan los correspondientes datos con el mismo método que hemos empleado para el resto de algoritmos y se entrenan los respectivos modelos. XGBoost no acepta dataframes de R, por lo que se han de convertir los dataframe a matrices.

```
train             <- readRDS("./datosPreGame/data_train")
test_preGame      <- readRDS("./datosPreGame/data_test")
target            <- readRDS("./datosPreGame/labels_train")
test_completo     <- readRDS("./datosPreGame/data_test")

predictors <- data.matrix(train)
rm(train)
```

Los parámetros escogidos son los siguientes

```
depth <- 13
eta <- 0.2
subsample <- 0.5
colsample <- 0.9
```

y procedemos a entrenar el modelo

```
xgboost_1 <- xgboost(data = predictors,
                             nfold = 10,
                             early_stopping_rounds =  30,
                             label = as.matrix(target),
                             num_class = 2,
                             max_depth = depth,
```

```
                              eta = eta,
                              nthread = 12,
                              subsample = subsample,
                              colsample_bytree = colsample,
                              min_child_weight = 1,
                              nrounds = 600, # PARA EL TESTEO LO HAS BAJADO A 20. ESTABA A 600
                              objective = "multi:softprob",
                              maximize = FALSE)
```

eliminamos del entorno de trabajo los predictores y el dataset de testeo para importar el dataset completo

```
 rm(predictors, data_test)

        train              <- readRDS("./datosCompleto/data_train")
        test_completo      <- readRDS("./datosCompleto/data_test")


        predictors <- data.matrix(train)
        rm(train)
```

Entrenamos el segundo modelo manteniendo los mismos parámetros

```
xgboost_2 <- xgboost(data = predictors,
                              nfold = 10,
                              early_stopping_rounds =  30,
                              label = as.matrix(target),
                              num_class = 2,
                              max_depth = depth,
                              eta = eta,
                              nthread = 12,
                              subsample = subsample,
                              colsample_bytree = colsample,
                              min_child_weight = 1,
                              nrounds = 600, # PARA EL TESTEO LO HAS BAJADO A 20. ESTABA A 600
                              objective = "multi:softprob",
                              maximize = FALSE)
```

Una vez tenemos ambos modelos entrenados procedemos a realizar las predicciones respectivas sobre el data_test y a generar las matrices de confusión correspondientes pasando las predicciones previamente por la función "Aciertos_a_confusión"

```
predicciones_completo <- predict(xgboost_1,newdata = data.matrix(test_completo))

results_1 <- data.frame(predicciones_completo[(1:length(predicciones_completo)) %% 2 == 0],
                        predicciones_completo[(1:length(predicciones_completo)) %% 2 == 1])
results_1 <- cbind(labels, results_1)
probs1    <- Aciertos_a_confusion(results_1)

matriz_confusion_1    <- confusionMatrix(data = as.factor(probs1[,4]), reference = probs1[,1])
predicciones_preGame  <- predict(xgboost_2,newdata =  as.matrix(test_preGame))

results_2 <- data.frame(predicciones_preGame[(1:length(predicciones_preGame)) %% 2 == 0],
```

```
                        predicciones_preGame[(1:length(predicciones_preGame)) %% 2 == 1])
results_2 <- cbind(labels, results_2)
probs2    <- Aciertos_a_confusion(results_2)

matriz_confusion_2 <- confusionMatrix(data = as.factor(probs2[,4]), reference = probs2[,1])
```

Por último, en el caso de XGBoost se ha de realizar el cálculo del RMSE manualmente

```
residuals_preGame  <- (as.numeric(labels) - 1) - as.numeric(probs2[,4])
RMSE_preGame       <- sqrt(mean(residuals^2))

residuals_completo <-(as.numeric(labels) - 1) - as.numeric(probs1[,4])
RMSE_completo      <- sqrt(mean(residuals_completo^2))
```