

Memoria Aprendizaje Supervisado

Joaquin Negrete Saab - Pablo Suárez Iglesias

Dataset

Para el trabajo, utilizamos el dataset de kaggle de [partidas de League of Legends](#) (LoL), un videojuego popular alrededor del mundo. El dataset incluye datos sobre varios aspectos de una partida antes de los 15 minutos, y tiene una variable respuesta binaria que indica si el equipo azul ganó esa partida.

Nuestro objetivo es entrenar un modelo mediante aprendizaje supervisado que pueda predecir qué equipo gana una partida de LoL con la información de los primeros 15 minutos de la partida.

El dataset traía información numérica de distintos aspectos de la partida de ambos equipos. Debido a que en una partida de LoL los valores absolutos no son lo más importante, modificamos el dataset para incluir las diferencias de estos valores numéricos. Y añadimos la suma de 3 variables que añaden contexto a la partida. Por ejemplo, la suma del oro total nos dice qué tan rápido pueden comprar los equipos. Eso ayuda a reducir la colinealidad y a reducir la dimensionalidad.

De esta forma, el dataset lo hemos separado en 3 datasets distintos, que se adaptan mejor a la forma de entrenamiento de los distintos modelos que utilizaremos para entrenar. El primero y el más completo tiene todas las variables, los valores absolutos y las diferencias de cada cosa. Este lo usamos para los árboles de decisión, ya que a estos no les importa la multicolinealidad.

Los otros dos tienen únicamente las diferencias y sumas de las variables. El tercero se diferencia del segundo porque no tienen las variables de suma y diferencia de oro, para ver cómo se comportaron los otros modelos si quitamos esto, que añade colinealidad con respecto a otras variables.

Escalamos con RobustScaler para no perder información sobre datos atípicos, que son partidas “stomp” e importantes para el modelo.

Modelos individuales

Para todos los modelos hemos dividido los datos utilizando `train_test_split` con un `random state` 42. De esta manera, al dividir los datos un 60% en train, 20% en validación y 20% en test, ningún modelo verá nunca los datos de test a la hora de escoger el mejor modelo. Para los classification reports usaremos los datos de validación.

Además, hemos utilizado la librería `shap`, que calcula los shapley values para dar interpretabilidad a los modelos.

Árboles de Decisión

Para la validación cruzada utilizamos el siguiente `param_grid`:

```
param_grid = {
    "criterion": ["gini", "entropy"],
    "max_depth": [5, 10],
    "min_samples_leaf": [5, 10, 20]
}
```

Donde los mejores parámetros fueron:
 {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 5}

Utilizamos estos criterios para ver cuál de los dos iba mejor. Luego estas profundidades máximas porque sabíamos que no iban a ser más de 10 niveles, y esos mínimos de datos en las hojas para evitar overfitting.

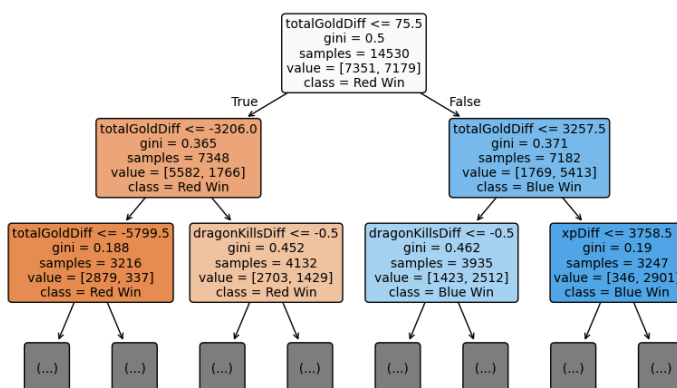
En nuestro árbol de decisión obtuvimos los siguientes resultados:

clase	precisión	recall	f1	support
Red Team Win	0.73	0.78	0.75	2435
Bue Team Win	0.76	0.71	0.73	2409
accuracy			0.75	4844

Podemos ver que tenemos un accuracy del 75%.
 Las features más importantes fueron la diferencia de oro, los dragones obtenidos, y la xp obtenida.

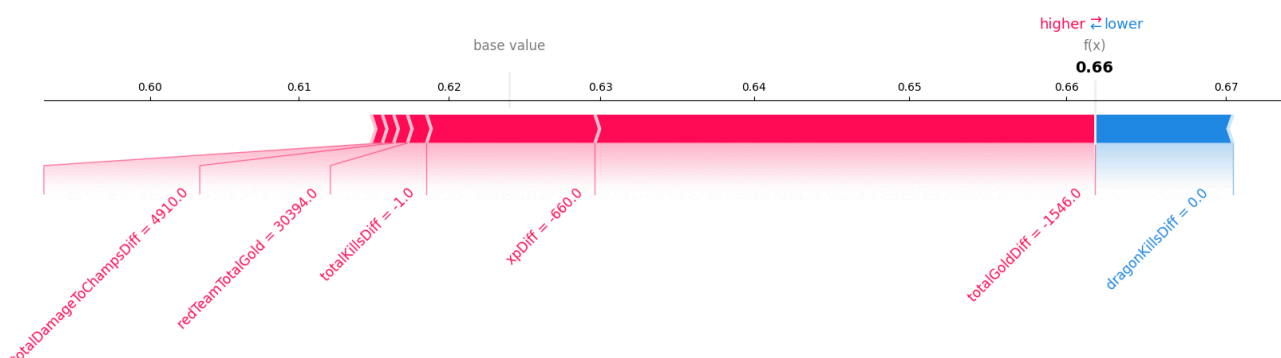
Feature	Importance
totalGoldDiff	0.906648
dragonKillsDiff	0.047862
xpDiff	0.027521

Visualización de las primeras reglas del Árbol de Decisión



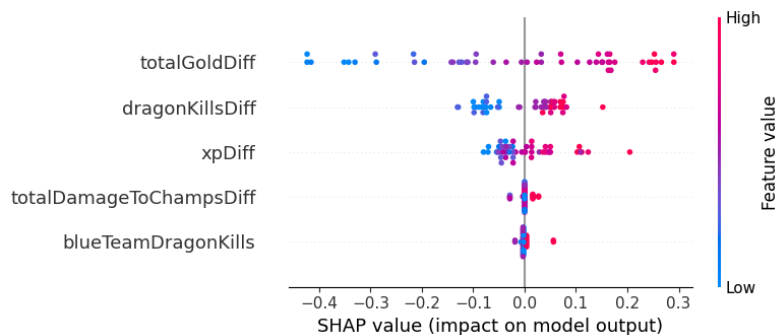
La regla raíz es totalGoldDiff <= 75.5. Podemos ver que efectivamente la diferencia de oro es la protagonista en este árbol, confirmando la tabla anterior. Todo esto tiene sentido, LoL es un juego gobernado por el oro. Todo en una partida se reduce a eso, salvo ciertos casos en donde alguien comete algún error tonto, pero eso es menos probable en estos niveles de ELO.

Veamos los valores de shap de una partida para explicarlo aún más.



Este gráfico nos indica qué variables “empujan” hacia la derecha la probabilidad —en esta partida— de que gane el equipo rojo. Shap calcula que el equipo rojo tiene una probabilidad de ganar del 0.66 en esta partida. Interpretando el gráfico, los valores en rojo de diferencia son negativos, es decir, que el equipo rojo tuvo más oro, más xp, más kills. Luego el único valor que “frena” al modelo, es que ambos equipos han tenido la misma cantidad de dragones.

Veamos ahora los indicadores globales del modelo:



Podemos ver nuevamente que el totalGoldDiff es el más importante. Y podemos ver que valores bajos, es decir, valores negativos, que el oro del equipo rojo es mayor al oro del equipo azul, indican un Shap value menor. Esto se refiere a que mientras más bajo el goldDiff, más probable es que gane el equipo rojo (clase 0) y viceversa. Luego vemos que la siguiente es

dragonsKillDiff, pero hay grupos porque es una variable “discreta”. Mientras más dragones tiene el equipo azul, mayor es el SHAP value. Y más alta es la probabilidad que gane el equipo azul, y así con todas las variables.

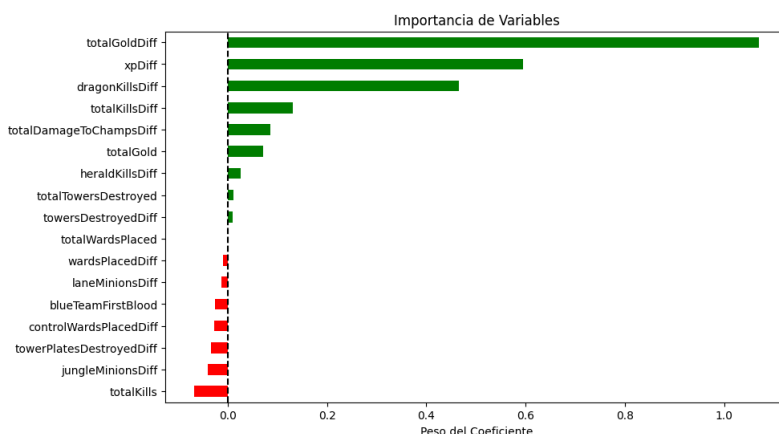
SVM

La validación cruzada de SVM tarda mucho por su gran carga computacional por lo que intentamos escoger pocos parámetros:

```
param_grid = [
    {
        'kernel': ['linear', 'rbf', 'sigmoid'],
        'C': [0.1, 1, 10]
    },
    {
        'kernel': ['poly'],
        'C': [0.1, 1, 10],
        'degree': [2, 5]
    }
]
```

Hemos separado por una lado el modelo polinomial pues tiene un parámetro extra que es el grado del polinomio. Si juntáramos ese parámetro con el resto, seguiría funcionando pues los demás kernels pueden recibir el parámetro degree, pero lo ignoran y entonces estaríamos haciendo entrenamientos repetidos del mismo modelo. El resultado de nuestro GridSearch es un modelo con los parámetros:

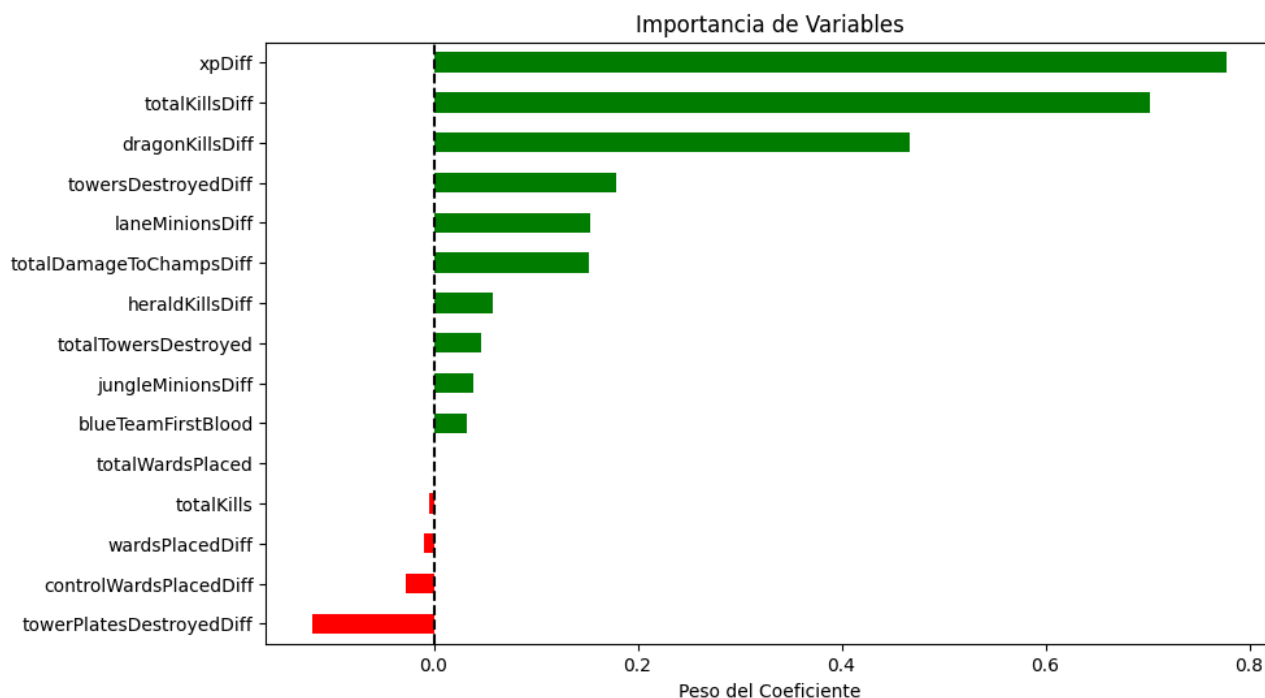
- Kernel : lineal
- C: 0.1



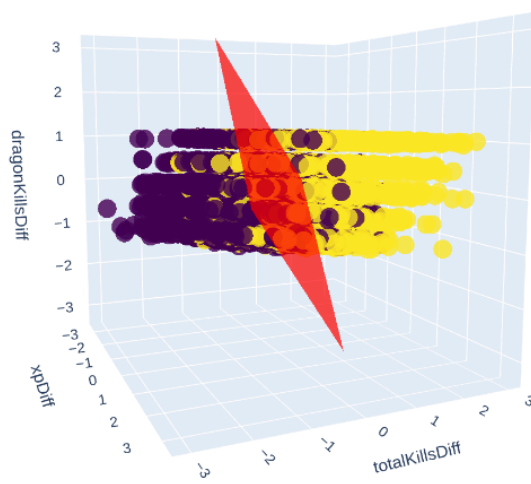
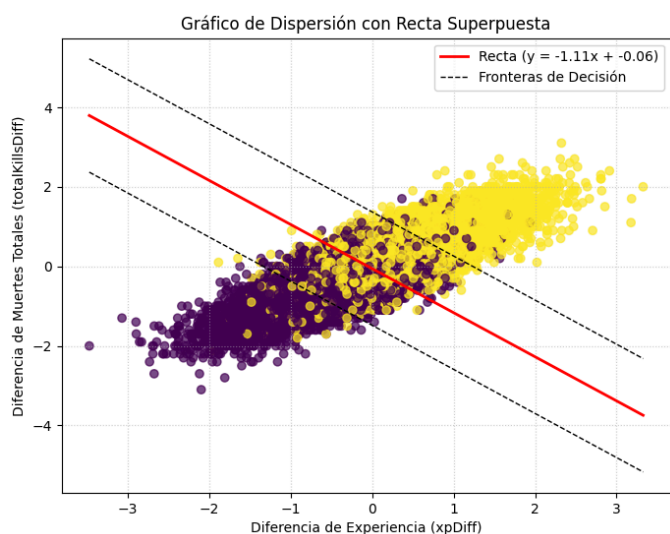
Estos parámetros nos indican ciertas conclusiones sobre nuestros datos. Siguen una relación directa, no hay islas o curvas raras a las que adoptar el modelo (de ahí que se escoja el modelo lineal). Sin embargo, una C baja significa que usa una alta regularización lo cual, en modelos como este, significa que hay mucho ruido o que nuestros datos se solapan bastante por lo que no son fáciles de separar espacialmente.

Después, procedemos a intentar con el dataset sin el oro. Esta vez tiene como resultado:

- Kernel: lineal
- C: 1



Esto confirma nuestras sospechas, la variable oro, aunque parezca la más importante, tiene una gran multicolinealidad con otras variables. Aunque esto parezca inofensivo, en modelos como este hace que una variable tome mucho protagonismo mientras que aquellas con las que tiene colinealidad tomen valores negativos o más bajos (equilibrando a esta variable), y por ello para evitar el overfitting el GridSearch escoge un C bajo, para aumentar la regularización. Este fenómeno se conoce como inflación de varianza. En ambos modelos hemos conseguido un accuracy de 0.76, lo cual añade a la conclusión de que, aunque los modelos cojan a la cantidad de oro como una variable importante, en verdad no añade información nueva. Para hacer que nuestros datos sean más significativos y reducir el overfitting procederemos sin el oro.



Para ver la separación hemos usado las variables que el modelo considera las más importantes, parece que la separación es bastante decente pero se aprecia que hay bastante solapamiento.

Regresión Logística

Este modelo, al ser muy simple, tiene una rejilla de parámetros muy sencilla:

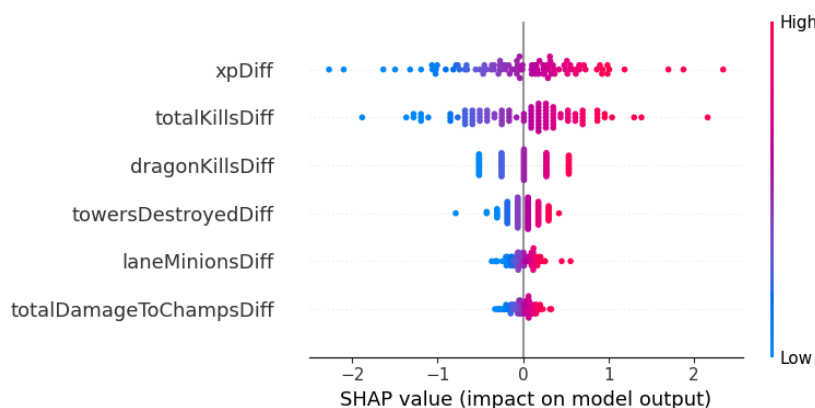
```
"logreg__C": list(range(1, 11))
```

Hemos escogido estos valores altos ya que sabemos por el modelo anterior que ya que el dataset sin oro tiene mucha menos multicolinealidad, el parámetro será alto. Y efectivamente, el mejor valor fue $C = 5$.

El reporte que tenemos de este modelo es el siguiente:

clase	precisión	recall	f1	support
Red Team Win	0.75	0.75	0.75	2435
Bue Team Win	0.74	0.74	0.74	2409
accuracy			0.75	4844

Y ahora veamos los shapley values globales:



Esto es un hallazgo muy interesante. Nosotros ya sabemos que el oro gana las partidas. También sabemos que los objetivos del juego recompensan al jugador con oro, pero también con otras cosas. Tirar torres mete presión al rival. Hacer una kill no solo te da oro a ti, pero le quita experiencia y oro al rival, por estar en respawn. Los dragones

además de oro dan pasivas que duran toda la partida. Entonces, al quitar el oro de las variables, podemos ver exactamente qué es lo importante en un juego. Y vemos que lo más importante es la experiencia (era de esperarse), pero también las kills. Pensamos que es precisamente por eso que explicamos antes. Los dragones también son importantes. Por eso nos gusta este modelo, porque nos explica *qué* elementos que dan oro son los más importantes.

KNN

La rejilla de hiperparámetros de este modelo es más compleja, ya que hay más variables qué escoger:

```
param_grid = {
    "knn__n_neighbors": list(range(k-21, k+11, 2)),
    "knn__weights": ['uniform', 'distance'],
    "knn__metric": ['euclidean', 'manhattan']
}
```

Donde $k = \sqrt{n}$ y n es el número de datos

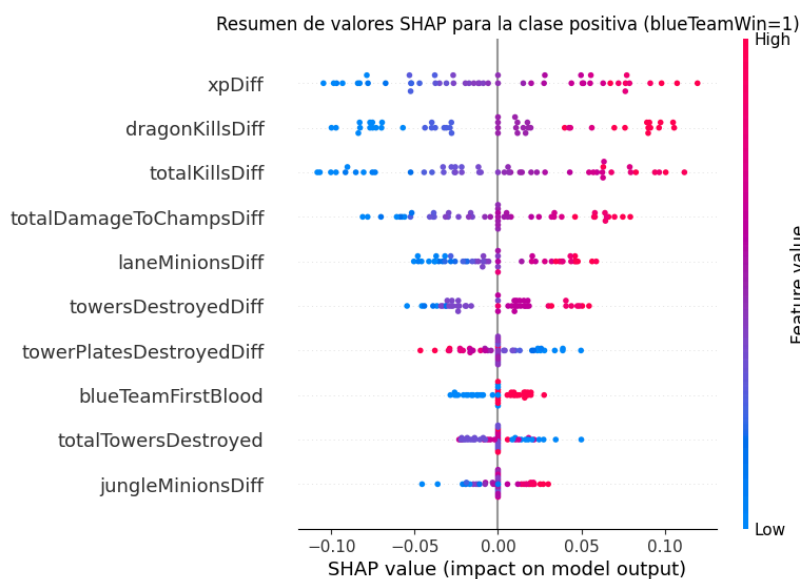
Mejores hiperparámetros: {'knn__metric': 'manhattan', 'knn__n_neighbors': 101, 'knn__weights': 'distance'}

Escogimos estos neighbors porque al haber muchos datos (más de 24000 entradas) y al ser las partidas de un mismo elo (esto hará que sean más parecidas unas de otras) pensamos que tendrían que haber muchos vecinos parecidos para clasificar el modelo.

Hemos obtenido el siguiente reporte de clasificación:

clase	precisión	recall	f1	support
Red Team Win	0.76	0.78	0.77	2449
Bue Team Win	0.77	0.75	0.76	2395
accuracy			0.76	4844

Veamos los valores shap para interpretar el modelo:



Podemos ver que, a diferencia de antes, ahora el modelo tiene muchas variables poco significativas (viendo el eje x, el impacto de las variables es muy bajo). Pero vemos que en este modelo le dan la misma importancia relativa a la experiencia, dragones y kills, y luego daño, minions de carril, torres...

Los valores relativos tienen sentido, es lo que hemos estado viendo en los modelos anteriores, pero sus valores absolutos no nos gustan porque SHAP nos está diciendo que ninguna variable tiene un especial impacto, cuando debería comportarse más como en

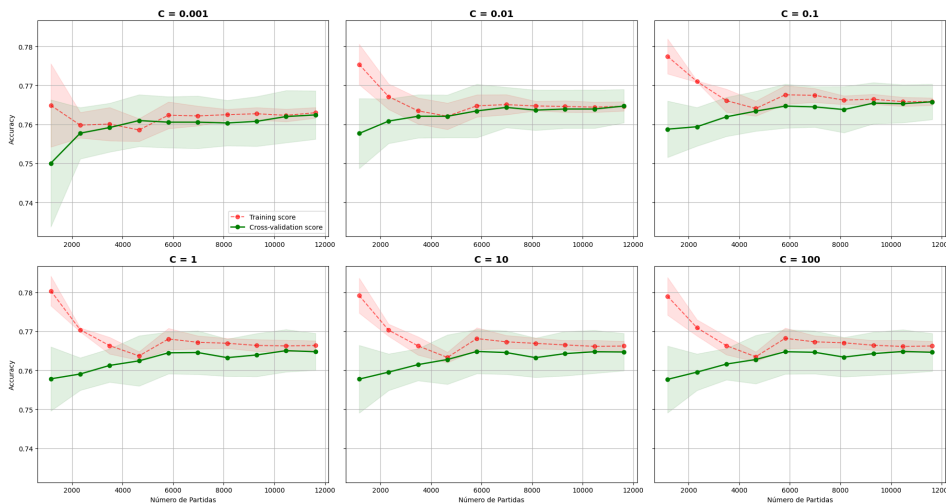
regresión logística, además de ser muy costoso computacionalmente.

¿Se puede mejorar el Accuracy?

Por ahora en todos los modelos que hemos usado hemos encontrado un accuracy de 75-76%. Además, a la hora de usar GridSearchCV, los modelos escogidos son bastante simples. Esto nos lleva a pensar que tenemos unos datos muy básicos, que siguen una relación lineal simple pero que contienen mucho ruido o están incompletos. Parece que los propios datos no son capaces de explicar lo suficiente para determinar con exactitud un ganador. Por esto, nos encontramos con un techo en la posible precisión. Creemos que no está limitada esta precisión ni por la cantidad de datos (la cantidad de datos con la que contamos es muy alta), ni por la simplicidad de los modelos escogidos. Para ver esto utilizamos varios métodos.

Primero, queremos ver la curva de aprendizaje de uno de nuestros modelos, cómo puede ser el de regresión logística (por ser un modelo con pocos parámetros):

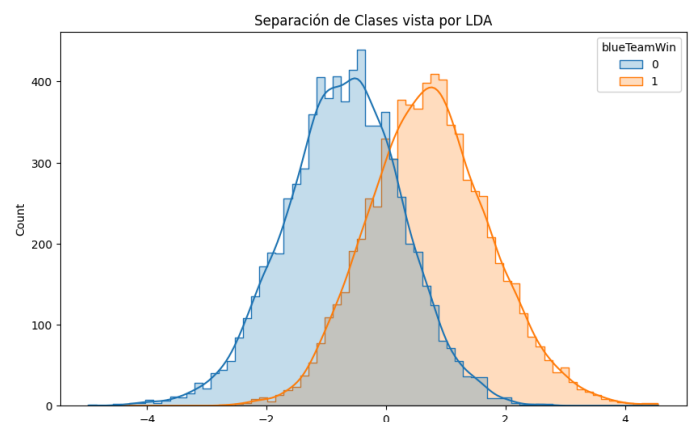
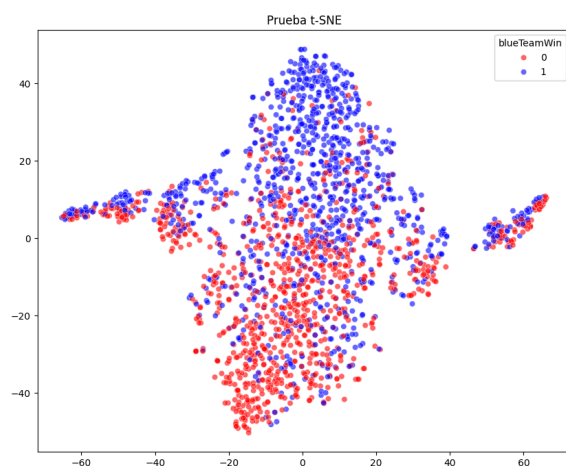
Impacto de la Regularización (C) en las Curvas de Aprendizaje



Vemos que, sin importar el parámetro escogido, las líneas se aplanan y se acercan mucho. Esto significa que el modelo ya ha conseguido aprender todo lo que puede, por lo que tener más instancias no ayuda.

Después intentamos usar un modelo más complejo para ver si este funcionaba mejor con los datos, para ver si simplemente es que nuestros modelos son demasiado simples, para ello usaremos un random forest. Usando GridSearch conseguimos un Random Forest con 500 predictores y, de nuevo, un 76% de precisión. Esto confirma que no es un problema de complejidad de nuestro modelo.

Un par de pruebas más que utilizamos son el t-SNE y LDA para ver si existe solapamiento:



Además hicimos lo siguiente: entrenamos un modelo Knn con $n_neighbors = 5$. Después para cada valor de X_test buscamos sus 5 elementos más cerca de este último, la predicción será 0 o 1, el que sea mayoría. Si no coincide lo contamos como confundido pues significa que está rodeado de valores distintos a sí mismo en su mayoría. Después calculamos el porcentaje. Hemos obtenido como resultado un 29.18%. Todos esos datos están rodeados de casos contrarios, esto es consecuencia del gran solapamiento de los datos.

Consideramos que el *accuracy* de aproximadamente ~75% obtenido por todos los modelos es un buen resultado dada a las restricciones temporales del problema.

Este dataset realiza predicciones basándose en los primeros 15 minutos de la partida, y de acuerdo a [esta](#) página, la duración media de una partida de este ELO es de ~25 minutos. Esto representa ~40% de tiempo extra que el dataset **no** puede ver.

Interpretamos este ~75% como el techo de predictibilidad propio del modelo en sí gracias al “Snowball effect” que tienen las partidas de LoL: el estado de la partida al minuto 15 determina el resultado en tres de cada cuatro casos debido a la acumulación de ventajas (las que hemos visto, oro, experiencia, dragones). Hemos demostrado que el ~25% de error restante no se debe a una deficiencia de los modelos ni de los datos, sino a la varianza no observada de la fase final del juego, y esto es porque pueden suceder situaciones como que hayan campeones que escalan muy bien, errores humanos o robos de objetivos, que pueden darle la vuelta a una partida. En conclusión, nuestros modelos han logrado capturar la inercia de la partida, llegando al límite de lo que es posible predecir sin la información sobre la composición de los campeones o eventos futuros.

Mejor modelo

Debido a que el accuracy es igual en todos los modelos, podemos saltarnos esta métrica e ir directamente a lo importante: explicabilidad, información aportada y eficiencia. Por estos tres motivos, hemos decidido tomar la regresión logística como el mejor modelo.

Podríamos haber tomado el árbol de decisión como mejor modelo, pero nos ha gustado más la regresión logística porque al quitarle el oro, nos explica a detalle cuánto aporta cada variable que a su vez da oro, por lo tanto, son igual de explicables pero la información aportada por la regresión es mejor. En cuanto a eficiencia, es ligeramente más fácil calcular un árbol de decisión.

SVM también nos ha gustado porque si de por sí es fácil de explicar, un SVM lineal aún más, pero es cierto que no es lo mejor computacionalmente hablando.

Por otro lado, KNN es muy ineficiente computacionalmente, y poco explicable.

Por estos motivos, hemos decidido utilizar la regresión logística.

A continuación el reporte de clasificación de test de regresión logística:

clase	precisión	recall	f1	support
Red Team Win	0.76	0.76	0.76	2455
Bue Team Win	0.76	0.75	0.75	2389
accuracy			0.76	4844

Vemos que tenemos un accuracy de 0.76, que es lo esperado gracias a todo lo explicado anteriormente, y que hemos entrenado satisfactoriamente el modelo.

Ideas para mejorar el modelo

Como ya hemos explicado antes, el modelo sufre de la característica temporal de los datos.

Ciertamente el modelo mejoraría si tuviéramos datos durante más tiempo, pero eso sería eliminar la problemática que queremos resolver: predecir el resultado de una partida antes de que acabe.

Por lo tanto, creemos que lo mejor para mejorar el modelo sería incluir dos cosas: la composición de campeones de cada equipo, así el modelo podría predecir que cuando hay ciertos campeones que tienen mal *early game* pero muy buen *late game* es más probable que la partida se gane; y con esto, deberíamos tener más datos de partidas, es decir, más filas en el dataset.

Incluso, tal vez eliminaríamos las variables de contexto, las que son suma de ambos equipos, para reducir aún más la dimensionalidad, ya que hemos visto que en ningún modelo aportan mucha información.