CHURN PREDICTION CHALLENGE

Informe Final

GASET, CONSTANZA
LOPEZ WALLACE, LUCIA MARIA
TISSONE, JUAN AUGUSTO

El siguiente informe pretende predecir si los usuarios de *Castle Crush* seguirán jugando luego del tercer día de haber instalado la app. Aquellos jugadores que luego del tercer día no hayan jugado más al juego serán considerado jugadores que hicieron *churn*. Como criterio para que sea considerado *churn*, el último día que jugaron debe ser el 3er día posterior a la instalación de la aplicación y luego no se repite la acción de jugar en los siguientes 14 días (del 4 al 17 inclusive).

El trabajo aquí descrito se encuentra divido en 5 partes:

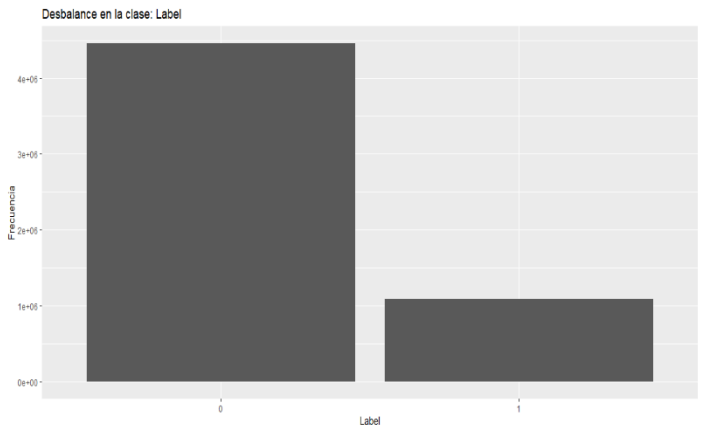
1. Análisis exploratorio de los datos
2. Selección e ingeniería de variables
3. Sistema de validación y modelo para la predicción de *churn*
4. Explicación de la estructura del código
5. Temporalización de las tareas

1. Análisis exploratorio de los datos

Los datos otorgados por la empresa creadora del juego *Castle Crush* se compone de un dataset con las siguientes características:

- Peso total 1.74GB (divididos en 5 archivos)
- Variables que componen el *dataset*: 101. Contiene variables tanto continuas como categóricas.

Para trabajar se tomó un total del 70% de los datos de entrenamiento (*train*) y el 100% de los datos de evaluación. La cantidad de datos de *train* fue de menos del 100% dado que trabajar con el 100% exigía mucho a las computadoras con las cuales se desarrolló el trabajo sin generar un beneficio equivalente al esfuerzo extra en términos de performance. Como primera medida se creó la variable *Label*, siendo 1 a aquellos datos catalogados como *churn* según el criterio antes descrito y 0 caso contrario. Luego evaluamos qué tan balanceado estaba el *dataset* en cuanto a esta variable.




```
        0         1
0.8036342 0.1963658
```

Como se ve en el gráfico anterior, el *dataset* se encuentra desbalanceado, habiendo un 80% de observaciones correspondientes a no c*hurn* mientras que el 20% corresponden a observaciones de usuarios que hicieron c*hurn* bajo los criterios de c*hurn* citados. Esto no afectará a la performance del modelo dado que la métrica seleccionada, AUC, no se ve afectada por datos desbalanceados. Luego, se analizó si existían variables que tuviesen observaciones duplicadas. Sin embargo, no se encuentran valores duplicados por lo que no se eliminaron registros del *dataset* por este motivo.

Después se analizaron la cantidad de NA que se encuentran en cada una de las variables. Se aplicó un criterio por el cual se desestimaron aquellas variables donde el 70% o más de las observaciones sean NA. De este análisis se descartaron las siguientes variables:

	na_prop
age	0.9699992
site	0.7299943

Por lo tanto, las variables *age* y *site* no son tenidas en cuenta para el análisis. Esta decisión se basa en que el utilizar estas variables, debido a la cantidad de NA´s que presenta, puede ser perjudicial al modelo (sumado a que sería ineficiente ya que estaríamos trabajando con datos que no sumarían predicción y por lo tantos descartarlos también sumaría a aumentar la eficiencia del análisis).

2. Selección e ingeniería de variables

Para el análisis de variables se utilizaron los archivos del *train set* con un *install date* < 383. Esto se debe a que datos posteriores contendrán información del futuro, lo que no hace confiable el valor de la variable *Label*.

Antes de comenzar con el análisis se separaron los sets de *evaluation, train y validation*. Esto se realizó con el objetivo de no caer en problemas de *data leakage* (principalmente por el cálculo de la mediana y las regresiones que se harán posteriormente).

Para nuestra selección de variables, en primer lugar, se analizó la varianza de las variables con el objetivo de descartar aquellas que tengan poca varianza (< 1); las cuales no van a aportar mucha explicación al comportamiento de nuestra variable *churn*, por el hecho de que la misma se mantiene constante o casi constante a lo largo de todo el *dataset*. Encontramos que la variable *traffic_type* como única variable con varianza < 1 y la eliminamos del *dataset*.

A continuación, detallamo las estrategias utilizadas para la selección de variables de nuestro modelo:

- Filtering: Observando el poder predictivo de las variables al correr una regresión contra *Label.*
- Embedded methods: Observando que variables fueron importantes para nuestro algoritmo.

Analizamos el grupo de variables:

Sum_dsi_X features

Encontramos 18 variables evaluadas en los días [0 ; 3].

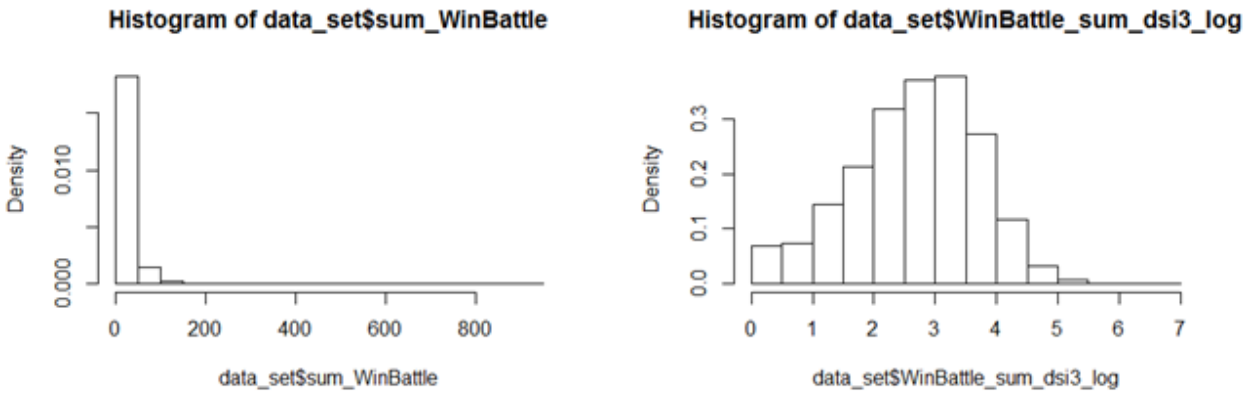
Primero, miramos cómo se distribuían y reemplazamos por la mediana los valores NA. Cabe destacar, que se utilizó la mediana porque notamos que muchas variables presentan valores extremos y estos datos sesgan la media; es por eso, que nos pareció mucho más robusto reemplazar por el dato de mediana en lugar de utilizar la media, medida que se ve afectada mucho por la presencia de *outliers*.

Analizamos si sumar el número de eventos en cada uno de los días aumentaba la explicación de nuestra variable a predecir; y encontramos que para algunos casos sí. En esos casos, tomamos la decisión de sumar las variables y dejar solo la nueva variable compuesta por la suma de las otras. No se conserva las variables que componen esta sumatoria ya que lo consideramos incorrecto debido a que puede aumentar de forma artificial el peso de esas variables en el modelo en el caso de conservar todo.

Para evaluar la significancia corrimos una regresión lineal contra *Label* y miramos el R^2 ajustado. Ejemplo, la variable *StartSession* aumenta explicación de *Label* al sumar los eventos.

Variable	R^2
StartSession_sum_dsi0	0.005607
StartSession_sum_dsi1	0.01284
StartSession_sum_dsi2	0.01433
StartSession_sum_dsi3	0.01663
Sum_StartSession	0.02252

Finalmente, notamos que la gran mayoría de estas variables tiene una distribución asimétrica positiva y presentan valores outliers. A continuación, mostramos como ejemplo cómo se comporta la variable sum_WinBattle; donde queda en evidencia que la mayoría de los datos se concentra en valores de [0,50]. Por lo que vamos a aplicar log, para suavizar este efecto.



Como aclaración se testeó la performance del modelo en test sin aplicar la transformación de log y se notó un incremento en la capacidad predictiva del modelo haciendo esta transformación por lo que se decidió hacer esto.

También se estudió de todos estos grupos de variables como estaban correlacionadas entre si, luego se observó que ningún par de variables de ninguno de los grupos estudiados poseía una correlación muy alta (mayor a 0,9. punto a partir del cual consideramos niveles muy altos de correlación).

Categorical features: Analizamos estas 7 variables categóricas. Todas ellas son factores con varios niveles. Ninguna de ellas presenta NA.

Other features: En esta sección se analizaron 7 variables. Descartamos la variable *user_id* porque que su poco poder predictivo.

Ingeniería de variables

Nos pareció interesante sumar al modelo las siguientes variables:

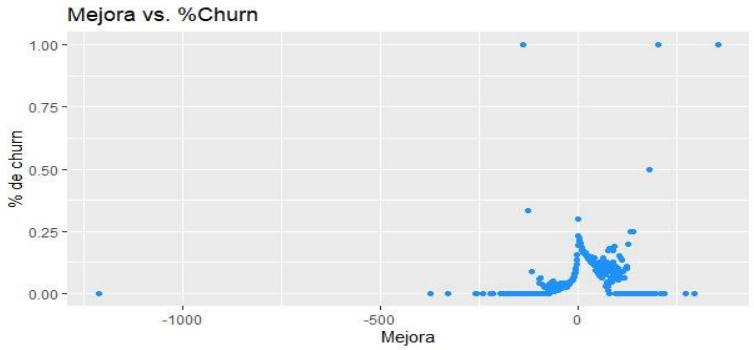
- Mejora

  Se busca entender la relación entre la mejora que tienen los jugadores a nivel de juego (esa mejora se mide como la diferencia entre batallas ganadas el día 0 y el día 3) y la tasa de *churn* que tienen.

A modo de aclaración:

Jugador que mejora = 'Batallas ganadas día 3' – 'Batallas ganadas día 0' > 0
Jugador que empeora = 'Batallas ganadas día 3' – 'Batallas ganadas Día 0' < 0
En el gráfico hay datos que se pueden considerar como outliers o que no tienen relevancia para el análisis; como por ejemplo, los valores negativos extremos que se pueden ver en el extremo izquierdo del gráfico. Estos valores representan a jugadores que jugaron más partidas al principio y después dejaron el juego (antes de día 3 y por lo tanto no son considerados como *churn* bajo la definición dada).

Si nos concentramos en la parte del gráfico correspondiente a los valores de mejora entre 0 y +50 (donde se concentra la mayoría de los datos) podemos ver una tendencia que muestra una baja en la tasa de *churn* cuánto mejor es la performance de los jugadores.



- Gasto

Creamos esta variable que suma el total gastado tanto en moneda hard y soft para entender si este comportamiento afecta el *churn*.

Como otro método de selección de variables se probó correr el modelo con un porcentaje menor del set de entrenamiento (20% del total de los datos de *train*) e identificar qué variables habían tenido mayor poder predictivo. Esto resultó en la siguiente lista de variables:

Variables seleccionadas			
PiggyBankModifiedPoints_sum_dsi1_log	OpenChest_sum_dsi0_log	install_date	sum_StartSession_log
PiggyBankModifiedPoints_sum_dsi2_log	OpenChest_sum_dsi1_log	LoseBattle_sum_dsi3_log	mejora
PiggyBankModifiedPoints_sum_dsi3_log	OpenChest_sum_dsi2_log	LoseBattle_sum_dsi1_log	EnterDeck_sum_dsi2_log
EnterShop_sum_dsi3_log	OpenChest_sum_dsi3_log	EnterShop_sum_dsi2_log	EnterDeck_sum_dsi3_log
sum_WinBattle_log	platform	train_sample	id
LoseBattle_sum_dsi0_log	EnterDeck_sum_dsi0_log	EnterShop_sum_dsi1_log	LoseBattle_sum_dsi2_log
sum_UpgradeCard_log	EnterDeck_sum_dsi1_log	EnterShop_sum_dsi0_log	

Luego, se corrió un modelo solo con estas variables identificadas esperando que la performance en test mejore, ya que se esperaba que remover el resto de las variables no tenga un efecto negativo en la performance, sino por el contrario que disminuya los niveles de "ruido" en los datos de entrenamiento y por lo tanto mejore la performance en test (a la vez que la velocidad de entrenamiento era mucho mayor).

3. Sistema de validación y modelo para la predicción de churn

El sistema de validación que elegimos en el presente trabajo es el método *validation/holdout set*. Este método consiste en separar una sub-muestra al azar de observaciones del *training set* como conjunto de validación. En nuestro caso,

dividimos de forma aleatoria el *training set* que contiene el 70% de los datos de *train* en dos conjuntos, uno de training (dtrain) con el 90% de los datos del training set original y otro de validación (dvalid) con el 10% de las observaciones del *training set* original. Con este método observamos que tan bien performa nuestro modelo en validación, buscando una performance alta (en nuestra métrica elegida en el modelo, AUC) evitando caer en *overfitting*.

Si bien los resultados obtenidos con este sistema de validación fueron buenos, posiblemente los resultados podrían haber mejorado si tomábamos como conjunto de validación los últimos 7 días del *dataset* de *train* (habiendo filtrado los últimos que tenían *data leakage* con el *dataset* de *test*, datos con *install_date* < 383). Creemos que este sistema mejoraría la performance del modelo por dos razones:

1. El set de validación, al ser más próximo en el tiempo, sería más parecido al set de evaluación.

2. Evitaríamos un posible *data leakage* borrando del *dataset* de entrenamiento los datos de los 14 días anteriores al primer día del set de validación (dada la definición de *churn*). Si no los borramos, esos 14 días sin jugar son un dato (que indirectamente está vinculado a la variable *Label_max_played_dsi* que determina el valor de la variable *Label*).

Lamentablemente no llegamos a probar esta alternativa antes de que la competencia finalizara.

El modelo que elegimos para el presente trabajo fue Árboles de Decisión, utilizando el proceso Boosting con el algoritmo de XGBoost. Este algoritmo de aprendizaje supervisado que intenta predecir de forma apropiada una variable de destino mediante la combinación de estimaciones de un conjunto de modelos más simples y más débiles. Construye los árboles de forma secuencial, utilizando información de los árboles previos, modelando en cada árbol aquello que no haya podido modelar o captar correctamente en los árboles previos, por lo que se considera un algoritmo de aprendizaje lento. Luego toma como predicción final una combinación de las predicciones de cada árbol, siendo esta la razón por la que se los considera modelos de ensamble.

XGBoost es una mejor formalización e implementación inteligente de Gradient Boosting Machine, que se aplica para modelos de regresión (también se pueden adaptar a modelos de clasificación). Se compone de 7 hiperparámetros que se deben definir previamente:

- **nrounds:** número de árboles.
- **max_depth:** máxima profundidad que pueden tener los árboles.
- **eta**: lo que conocemos como learning rate (velocidad de aprendizaje del modelo).
- **gamma:** es la mínima reducción del error para generar un corte.
- **colsample_bytree:** son las variables a muestrear y considerar en un árbol.
- **min_child_weight:** mínima cantidad de observaciones en los hijos para considerar un corte.
- **subsample:** muestreo de observaciones para considerar en cada árbol.

Resulta importante la definición de estos hiperparámetros para evitar que el modelo genere *overfitting* (ajuste demasiado los datos de *train*, incluso al ruido). En términos generales, algunos de ellos a medida que aumentan pueden generar *overfitting* (nrounds, max_depth, eta, colsample_bytree, subsample) mientras que otros a medida que decrecen pueden generar el mismo efecto (gamma, min_child_weight).

Como se mencionó, este modelo trabaja mejor con variables continuas, por lo que transformamos las variables de nuestro *dataset* aplicándole One-Hot-Encoding para transformarlas en matrices ralas. Este proceso transforma variables categóricas a variables continuas. A su vez, el uso de matrices ralas optimiza el uso de memoria RAM, cuestión que fue importante en el presente trabajo dado que el volumen de datos era elevado. Por otro lado, para encontrar los hiperparámetros óptimos utilizamos la búsqueda de random grid search, que optimiza la búsqueda de hiperparámetros mediante valores random (dentro de los límites establecidos en la función) que abarcan en gran medida el espacio de hiperparámetros explorando más valores para los hiperparámetros relevantes del modelo.

Se corrieron dos modelos, uno con las variables resultantes del análisis de variables y la ingeniería de atributos. Este modelo exploratorio contenía 58 variables y utilizó el 20% de los datos de entrenamiento y validación. Luego, de este modelo exploratorio se seleccionaron las variables más importantes y se corrió otro modelo, más simple, pero con las variables más significativas. Este modelo utilizó 27 variables (más la variable a predecir - *Label*) y el 70% de los datos.

Antes de seleccionar Árboles de Decisión con Boosting (XGBoost) como modelo, probamos otros modelos como Vecinos más cercanos y Árboles de Decisión. Dado que éstos no éstos no tuvieron mejor performance, decidimos avanzar con XGBoost.

Luego de tunear los hiperparámetros, nuestro mejor modelo obtuvo las siguientes características:

Nrounds:	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample
2	366	12	0.00129	0.82742	0.81264	4.34692	0.79043

Train-auc: 0.734054
Valid-auc: 0.720226

4. Explicación de la estructura del código

Se decidió darle al código una estructura modular, dividiéndolo en 3 partes:

1. Extract.r
2. Transform.r
3. Fit.r

Esta decisión se tomó ya que se consideró que de esta manera se iba a tener más facilidad a la hora de generar modificaciones al código con el fín de ir haciendo las distintas pruebas que se realizaron a lo largo del desarrollo del presente trabajo.

Cada una de los módulos cumplía las siguientes funciones:

Extract.r: Este módulo se encarga de la carga de los datos

Transform.r: Este módulo tiene como principal objetivo el análisis exploratorio y la ingeniería de variables. En este módulo también se determina los set de entrenamiento (t*rain_set*), validación(*val_set*) y evaluación (*eval_set*).

Fit.r: En este módulo se determina la grilla de hiper-parámetros para realizar el tunning, se entrena el modelo, se generan las predicciones y finalmente se guardan los resultados

5. Temporalización de las tareas

Análisis exploratorios de los datos	4 días
Ingeniería y selección de variables	7 días
Validación del modelo	3 días
Tunning de hiper-parámetros y selección del modelo	4 días
Armado y presentación del informe	2 días