Universidad ORT

Facultad de Ingeniería

Técnicas de Machine Learning para Análisis de Datos

Obligatorio 1

Alfredo Rodríguez
Marcelo Guelfi
Raúl Mayobre

Montevideo, Julio 2020

Indice

1 Introducción

El objetivo de este obligatorio fue predecir si un grupo de clientes abandonara una determinada empresa. En este caso es una empresa de telecomunicaciones. Para lograr el objetivo se debieron utilizar varias técnicas y algoritmos vistos durante el curso.

Se utilizaron como insumo dos *dataset*; uno para entrenar (*train*) y ajustar los modelos, y otro (test) para realizar la prueba final del modelo seleccionado.

El criterio para seleccionar y/o medir la performance de los modelos es mediante la utilidad.

1.1 Ingeniería de datos

Se evaluó realizar ingeniería de datos en el *dataset* al comenzar, de modo de reducir la cantidad de predictores a los efectos de disminuir el tiempo de ejecución y mejorar la utilidad. Si bien a priori en la exploración de los datos, se observaron variables predictoras que tenían mayor incidencia en la variable a predecir (*churn*) que otras, y que algunas variables podían estar correlacionadas, se optó por no reducir la dimensión de las variables predictoras.

Se intentaron utilizar algunos mecanismos de pre-procesamiento de datos, como PCA *(Principal Components Analysis)* y RFE *(Recursive Feature Elimination)* que están implementados en Caret. Sin embargo, los resultados no fueron concluyentes y se opto por no incluirlos.

El único pre-procesamiento en el *dataset* que se realiza es quitar los NA y eliminar la variable *ServiceArea* la cual presenta muchos valores. Para los algoritmos de *Random Forest*, se utiliza el comando *na.roughfix* (donde para valores numéricos, los NAs son sustituidos por las medianas de las columnas).

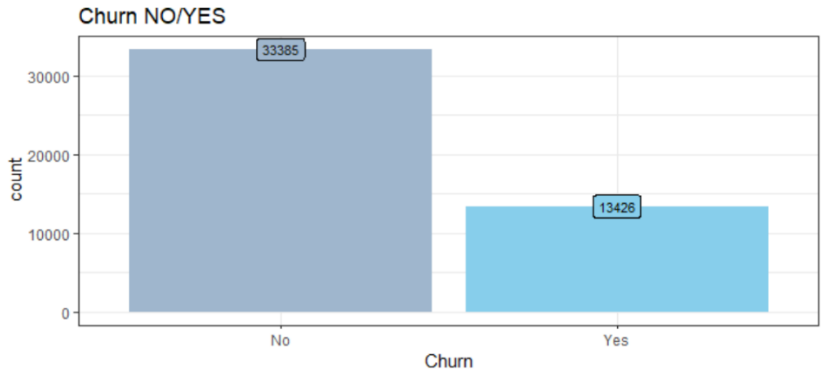
En el Anexo 1 se detalla el análisis realizado de algunos predictores, para luego intentar mejorar el rendimiento de los modelos, pero esto ultimo no se llevo a cabo.

1.2 Hold-out

El *hold-out* utilizado en el *dataset* se definió en 80% para *train* y 20% para *dev*. Con el 80% para *train*, se tiene un total de 37.448 observaciones, contra 55 variables (luego de hacer el pre-procesamiento indicado en el punto anterior). Para algunos casos también se usó un *hold-out* de 90/10. Luego para el entrenamiento en CARET, para la selección de los parámetros se usó la técnica de *Cross Validation* para maximizar la utilidad en *train,* sin sobreajustar.

1.3 Utilidad

La variable para predecir es *Churn*. En una primera exploración de *dataset* se puede observar que NO/YES se distribuyen en 71,3% y 28,68% respectivamente.



Para este caso, la utilidad o métrica definida es la siguiente:

- Cada falso positivo genera un costo de 10
- Cada positivo verdadero genera una utilidad de 25.5

Visto de otra forma → Utilidad = 25.5xTP – 10xFP

Llevando la utilidad en términos de distribución de *baseline*, es decir si no aplicamos técnicas de *machine learning* (ML de aquí en más) a los datos:

- **Utilidad mínima**, asumimos que todos los clientes van a rescindir el contrato, o sea *churn = YES;* 25.5 x 0.2868129 – 10 x 0.7131871 = **0.181858**
  O visto de otra forma: ((-10 x 33385) + (25.5 x 13426)) /46811 = **0.181858**
- **Utilidad máxima,** asumimos que acertamos a todos los clientes que van a rescindir el contrato *churn = YES*; 25.5 x 0.2868129 = **7.313729**
  O visto de otra forma; (25.5 x 13426) /46811 = **7.133729**

Por lo tanto, los algoritmos de ML a utilizar nos darán **utilidades** entre **0.181858 y 7.133729.** De acuerdo con lo informado por los docentes durante la elaboración del presente obligatorio, un valor de utilidad en el entorno de 2 se considera como aceptable.

1.4 Umbral de decisión

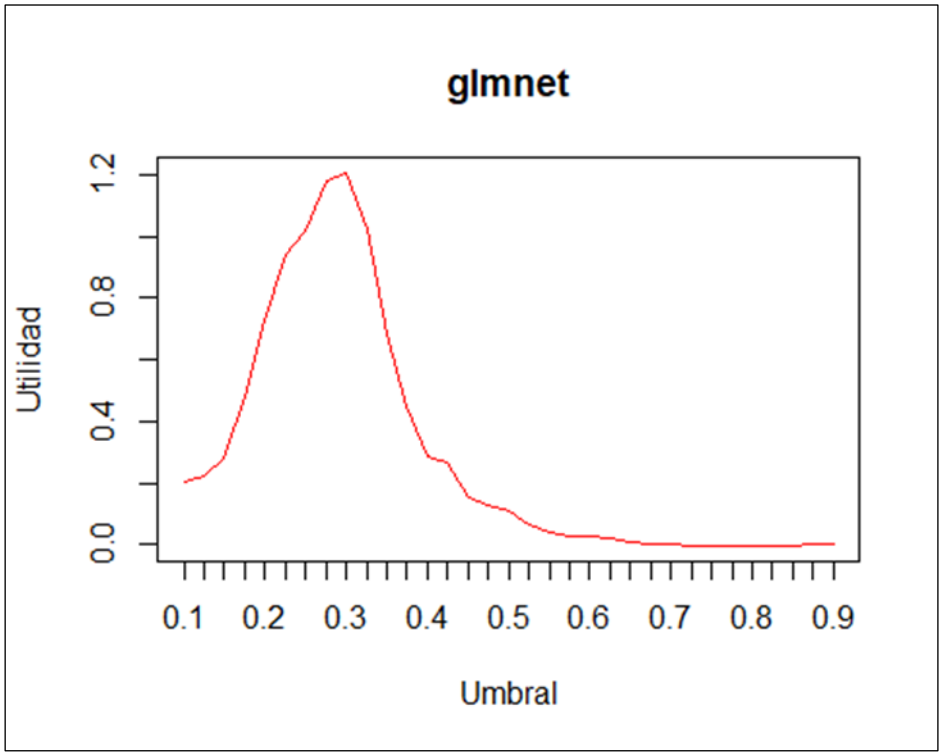
Como en todo método de clasificación, hay que establecer un umbral de decisión que indique si la observación o en este caso la predicción, pertenece a una clase o la otra (YES o NO).

Los clasificadores implementados en los algoritmos de clasificación, por defecto tienen establecidos este umbral en 0,5 (indicado en términos de probabilidad), que es el que usa el clasificador Bayesiano y que minimiza el error. Pero en este caso, que en la utilidad se penalizan los falsos positivos, este umbral no arroja buenos resultados, tal como se vio en los ensayos con *glmnet* con el script *glm.R.* Este algoritmo tiene establecido 0,5 como umbral de clasificación, y los resultados son muy bajos:

Utility en train CV: 0.094820311
Utility en dev: 0.1078295
Utility en train: 0.09928169

Por lo tanto, se debe encontrar el rango de umbral óptimo para cada algoritmo para maximizar la utilidad. Para luego ejecutar el script que calcula la hipótesis usada para la predicción. El rango de valores de umbral se define en el vector *df.thr_vec*. También se puede reducir el "paso" de incremento de valores para este vector. Al aumentar la resolución se puede encontrar el valor optimo del umbral.

Para el caso de glmnet, mediante el script *glm_prop.R* se obtiene el rango de valores de umbrales, donde se tendrá los mejores valores de utilidades en CV.



En el gráfico anterior se puede observar, que los mejores valores para el umbral estarán entre 0,2 y 0,4.

En CARET para poder usar métricas personalizadas, se usa el argumento *summaryFunction* de la función *trainControl.* Al tener que trabajar con valores de umbrales de probabilidades, hay que establecer *ClassProbs = TRUE* en *trainControl,* de modo que los resultados devuelvan las probabilidades y no la clase.

Otro enfoque al problema del umbral, es entrenar al algoritmo, usando la métrica ROC, la cual considera todos los valores posibles de umbral. Esto se logra en CARET con la función ya incorporada *twoClassSummary* para la *summaryFunction.*

1.5 Exploración del espacio de parámetros de ajuste

Además del espacio de valores de umbral mencionado en el punto anterior, para cada algoritmo se ensayaron varias pruebas para distintos valores de los parámetros de ajuste, que varían de acuerdo con cada algoritmo.

En CARET estos parámetros se pueden modificar con la función *trainControl,* donde también se establece el método de búsqueda de los parámetros. Este método puede ser *random* que implementa una selección aleatoria de combinaciones de parámetros de ajuste (la cantidad de combinaciones está dada por el valor *tunelength*), o *grid* donde se establece explícitamente el espacio de valores posibles para los parámetros de ajuste.

Hay algoritmos en los que el método *random* puede resultar beneficioso para encontrar valores razonables de los parámetros de ajuste en un tiempo relativamente corto. Sin embargo, hay algunos modelos donde la elección aleatoria de parámetros no arroja buenos resultados. Por lo tanto, es necesario hacer pruebas, ajustar, avanzar, retroceder hasta llegar al modelo final.

La metodología para la búsqueda de parámetros fue la siguiente:

- Se realizó una primera prueba con *random*, con valores de *tunelength* comenzando en 10 e incrementándose en 10 hasta no obtener una mejora en la utilidad de CV y dev.
- A partir de los parámetros encontrados en el punto anterior, se entrenó el algoritmo con el método *grid* en un espacio reducido de valores alrededor de los óptimos del punto anterior. También se evaluó la utilidad en *train* para chequear que los valores sean similares y comprobar que no estemos en la zona de sobre-ajuste.
- Luego, con las predicciones que obtuvimos mejores valores de utilidad en CV y dev, evaluamos su utilidad en test en Kaggle.

Nota: Se probó con la metodología Adaptive Resampling de CARET con el método random. Esta técnica (experimental) tiene la habilidad de re-muestrear en forma adaptativa los parámetros de modo de focalizarse en los valores próximos a los óptimos, evitando así muchos ajustes innecesarios y reduciendo el coste computacional. Sin embargo no se obtuvieron buenos resultados con esta técnica en cuanto a la utilidad.

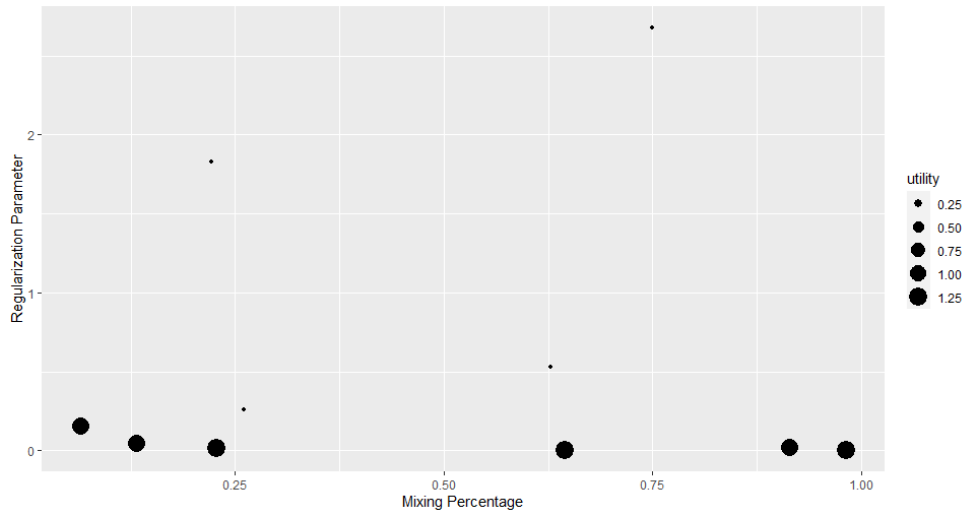
2 Algoritmos de Aprendizaje

2.1 GLMNET

Como ya se mencionó, este problema es del tipo de clasificación y no de regresión, por lo tanto es esperable no tener muy buenos resultados con esta técnica.

Parámetros a ajustar en GLMNET:

- ***Alpha;*** *Mixing Percentage*. Es el parámetro de penalidad entre 0 y 1. Cuando es 0 corresponde a la penalidad de Ridge y cuando es 1 corresponde a la de Lasso.
- ***Lambda****; Regularization Parameter*. Define una grilla para el valor de alpha. Se lo puede definir pero típicamente se deja que su valor sea establecido automáticamente por el programa entre lambda.max y lambda.min.ratio.



En la gráfica anterior se puede observar que los mejores valores de utilidad en *train* con CV, se logran para valores de lambda próximos a cero.

Aunque GLMNET tiene la ventaja de ser de ejecución rápida, los valores de utilidad obtenidos no fueron muy alentadores, en la siguiente tabla podemos observar las distintas utilidades obtenidas con distintos grupos de parámetros.

Algoritmo	df.form	Parámetros	Utilidad en CV	Utilidad en dev	Utilidad en train
glmnet	Churn ~ .	5 folds tunelength = 10 thr = entre 0,1 y 0,4 search = random	1.330919	1.305811	1.384443
glmnet	Churn ~ .	5 folds alpha = seq(0.5, 0.7, 0.01) lambda = seq(0.001, 0.003, 0.0001) thr = entre 0,1 y 0,4 search = grid	1.343536	1.308161	1.376699
glmnet	Churn ~ .	5 folds Adaptive Resampling thr = entre 0,1 y 0,4 search = random tunelength = 10	1.320906	1.279267	1.366872
glmnet	Churn ~ .	5 folds tunelength = 20 thr = entre 0,1 y 0,4 search = random	1.33606	1.290269	1.379009

Con el método *random*, subiendo el tunelength de 10 a 20 se empeoró el resultado. Tampoco mejoró con el método experimental “adaptive resampling” comentado anteriormente.
Los mejores resultados se obtuvieron con el método grid con alpha = 0.61 y lambda = 0.0015.

2.2 RPART

RPART es la implementación de R del algoritmo CART (Classification and Regression Trees). En este caso lo usaremos como un árbol de clasificación.

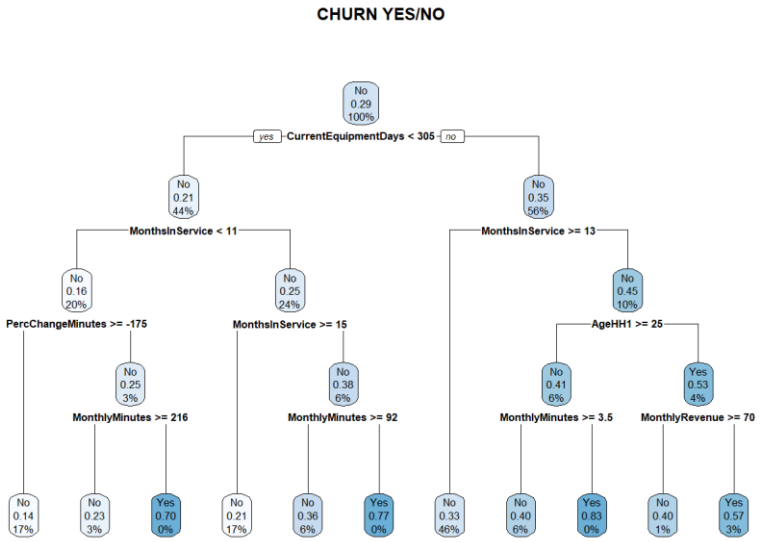
Parámetro a ajustar de RPART utilizando CARET*:*

> ***Complexity Parameter;*** CP. El objetivo es ver si un árbol más pequeño puede lograr resultados comparables al árbol entero, y así reducir la probabilidad de sobreajuste. La estrategia es recortar el árbol de modo de evitar *splitear* una partición si el *split* no mejora la calidad del modelo. Esto es controlado por este parámetro, que penaliza a los árboles que tienen muchos *splits*. El valor por defecto es 0.01. Cuanto mayor sea CP, más pequeño será el árbol. Un árbol pequeño puede ser más claro en términos de representatividad, pero puede ser no del todo bueno en cuanto a la performance del modelo.

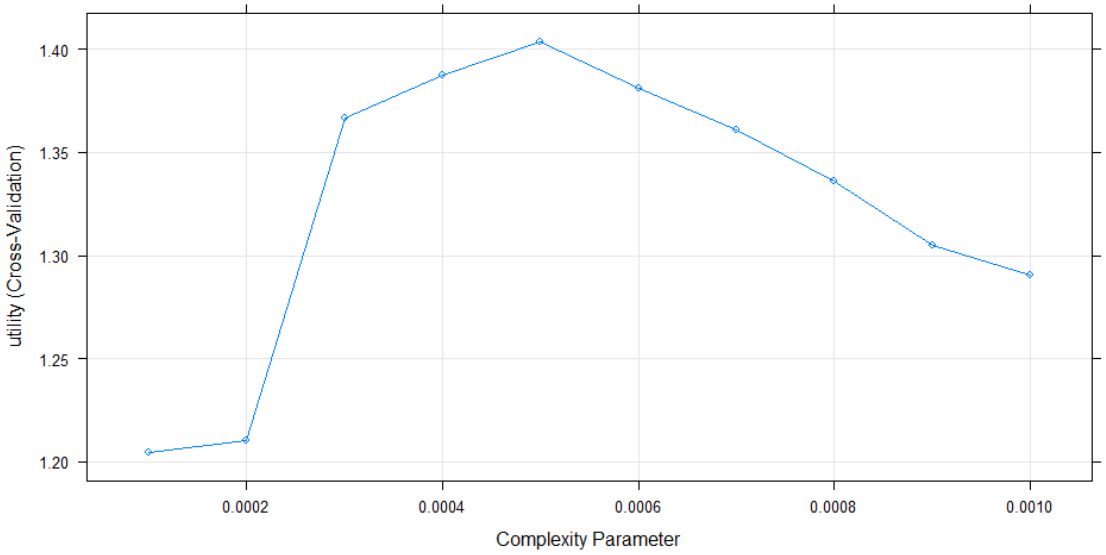
CP es el único parámetro que permite ajustar CARET de RPART*,* otros parámetros que se pueden ajustar si se utiliza RPART sin CARET son;

- ***Minsplit***; número mínimo de observaciones en un nodo para realizar una separación.
- ***Maxdepth***; máxima profundidad del árbol final.

Dentro de la documentación entregada como anexos hay un *compile report de R (script_rpart_final.html)* donde genero un árbol con *Maxdepth= 5*, con el fin de generar un árbol que sea agradable visualmente. El árbol obtenido es el siguiente;



Utilizando CARET se obtuvo la siguiente gráfica, donde se puede apreciar la evolución de la utilidad en CV, en función del parámetro CP del algoritmo RPART. Donde se puede apreciar que el valor optimo de CP esta entre 0.0004 y 0.0005. Donde el máximo se encuentra en 0.0004894649.



La siguiente tabla muestra el resultado de la utilidad en CV, dev y train variando los parámetros *tunelength*, CP y el modo de búsqueda (*random*) de CARET.

Algoritmo	df.form	Parámetros	Utilidad en CV	Utilidad en dev	Utilidad en train
rpart	Churn ~ .	5 folds tunelength = 10 thr = entre 0,1 y 0,3 search = random	1.381095	1.489906	1.560442
rpart	Churn ~ .	5 folds tunelength = 20 thr = entre 0,1 y 0,3 search = random	1.414086	1.493965	1.552244
rpart	Churn ~ .	5 folds tunelength = 30 thr = entre 0,1 y 0,3 search = random	1.414086	1.493965	1.552244
rpart	Churn ~ .	5 folds thr = entre 0,1 y 0,3 search = grid cp entre 0.00001 y 0.008	1.290719	1.381756	1.295709
rpart	Churn ~ .	5 folds thr = entre 0,1 y 0,3 search = grid cp entre 0.0001 y 0.001	1.403512	1.334544	1.350664

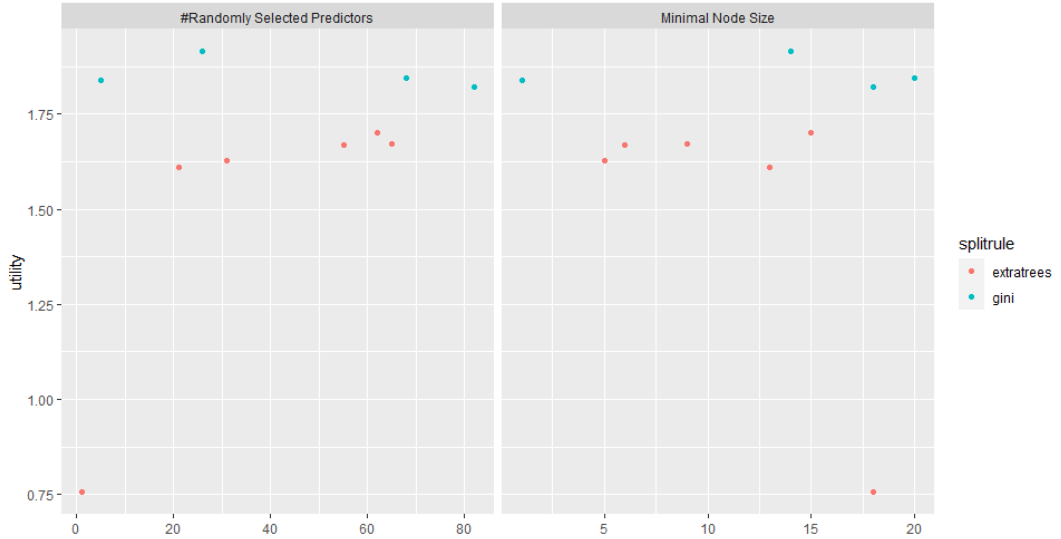
2.3 RANDOM FOREST

RANGER; es una implementación rápida del modelo de *bagging Random Forest*. En Caret solamente algunos parámetros pueden ser personalizados.
La cantidad de árboles está establecido en 500 (num.trees). Por lo general no es necesario un valor demasiado elevado para obtener buenos resultados.

Parámetros para ajustar en RANGER:

- ***mtry*;** cantidad de variables o predictores "m" que pueden elegirse de forma aleatoria para realizar un Split en un nodo. Justamente este número es la diferencia entre *random forest* y un modelo de *bagging* convencional. Cuando *mtry* es menor que "p" (con p = número de variables), se baja el nivel de correlación entre los distintos árboles generados, lo cual disminuye la varianza. Típicamente se elige m=√p para problemas de clasificación. Sin embargo, cuando existen pocos predictores relevantes, un valor más elevado de *mtry* puede producir mejores resultados, dado que aumenta la probabilidad de elegir los predictores más importantes.
- ***min.node.size*;** este parámetro regula la complejidad de los árboles. Es la cantidad mínima de *splits* que pueden realizarse en un nodo. Cuanto mayor sea este número, mayor complejidad en los árboles y menor profundidad. El valor por defecto es 1 para problemas de clasificación. Aumentando el *min.node.size* también se reduce el tiempo de ejecución del algoritmo.
- ***splitrule**;* define el criterio o regla de *split*. En el caso de clasificación, se define el *split* que minimice la impureza Gini, y en el caso de regresión la que minimice el RSS (*Residual Sum of Squares*). En CARET además están definidas otras reglas además de, GINI y RSS.

En la siguiente gráfica se puede observar la utilidad en CV al modificar algunos de los parámetros mencionados anteriormente (*mtry y min.node.size*), utilizando CARET con un entrenamiento en modo *random*.



En la siguiente tabla se pueden apreciar varios experimentos realizados con RANGER y CARET, estos experimentos son los que mejores utilidades entregaron.

Algoritmo	df.form	Parámetros	Utilidad en CV	Utilidad en dev	Utilidad en train
ranger	Churn ~ .	2 folds thr = entre 0,2 y 0,4 method = grid mtry = 9 splitrule = gini min.node.size = 4	1.849051	1.846771	7.353372
ranger	Churn ~ .	5 folds thr = entre 0,2 y 0,4 method = grid mtry = 9 splitrule = gini min.node.size = 9	1.87038	1.759271	7.315612
ranger	Churn ~ .	5 folds thr = entre 0,2 y 0,4 method = random tuneLength = 10	1.914336	1.816823	7.251549

Si bien los valores obtenidos de utilidad en CV y dev son mejores respecto RPART y GLMNET, en todos los casos podemos observar un sobreajuste. Esto nos lo indica la perdida de utilidad que se va dando desde *train* a CV y dev. En el caso de *train* (*), es muy abrupta la diferencia, pero también hay una perdida de utilidad en *test de* Kaggle que estuvo en el orden de 1,47.

(*) Nota: la utilidad de train que se obtuvo es excesivamente alta como se puede ver en la tabla anterior, incluso mayor a la utilidad de base o a priori. No pudimos encontrar una explicación a tan alto valor.

2.4 GBM

GBM (*Gradient Boosting Machine*) es un algoritmo de ensamble del tipo *boosting*. Difiere de *Random Forest* en que crea los árboles en forma secuencial en vez de en paralelo. En cada iteración utiliza el árbol previo para aprender y mejorar el modelo.

Los parámetros de ajuste de GBM que pueden personalizarse en CARET son:

- ***n.trees****;* es la cantidad de árboles que establece el algoritmo de *boosting*, o en forma equivalente, el número de iteraciones. Es decir, número de modelos que forman el ensemble. Cuanto mayor es este valor, más se reduce el error de entrenamiento, pero mayor riesgo de tener sobreajuste.
- ***interaction.depth****;* define la profundidad máxima de cada árbol en el ensemble. A mayores valores, mayor complejidad del modelo.
- ***shrinkage****;* este parámetro, también conocido como *learning rat*e o *step-size reduction*, controla la influencia que tiene cada modelo sobre el conjunto del ensemble. Es preferible mejorar un modelo mediante muchos pasos pequeños que mediante unos pocos grandes. Por esta razón, se recomienda emplear un valor de shrinkage tan pequeño como sea posible, teniendo en cuenta que, cuanto menor sea, mayor será el

número de iteraciones necesarias. Cuanto mas pequeño este valor, mayor la complejidad del modelo.

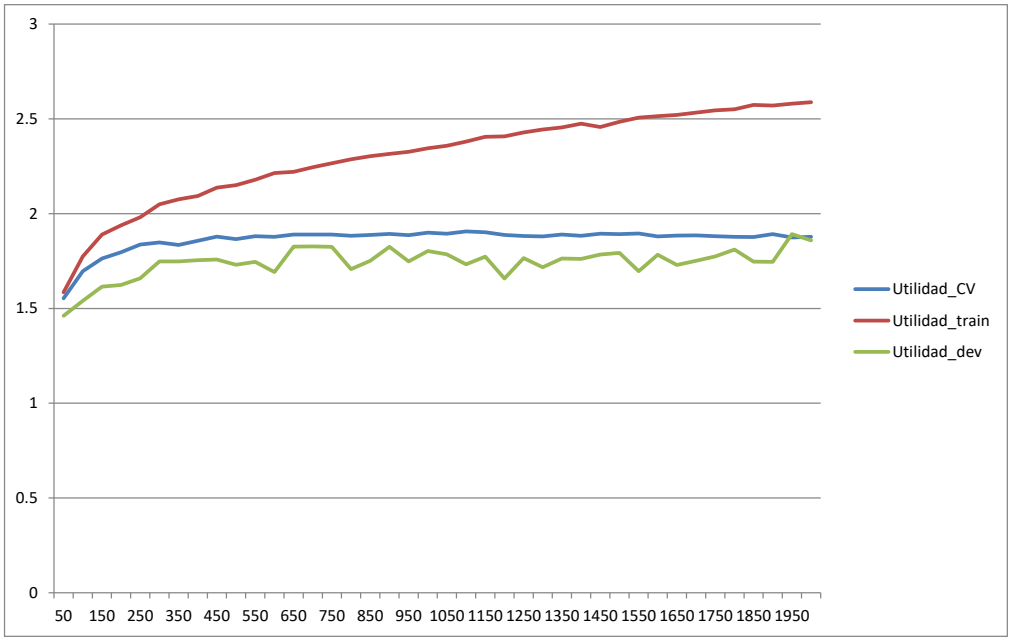
- ***n.minobsinnode****;* es un numero entero que especifica la cantidad mínima de observaciones en los nodos terminales de los árboles.

Para clasificar con este algoritmo se utilizó como base el código entregado por el docente, y se fueron variando los distintos parámetros que permite CARET. Las primeras pruebas se hicieron con la opción *Random*, dejando que CARET realice la selección de los valores.

El modelo entregado por CARET nos indicaba que en el entorno de 750 arboles eran necesarios para obtener los mejores valores de utilidad (mejores que GLMNET y RPART) en CV y *dev* (en el entorno de 1.8 a 1.9 de utilidad).

A partir de los valores óptimos que entrego CARET, se realizaron múltiples ajustes con el fin de seguir mejorando la utilidad. En algunos casos, el modelo que nos entregaba un buena utilidad en *train* con CV en CARET, luego en *dev y* Kaggle no daba tan buenos resultados. Por este motivo se modificó el script de base para poder calcular con cada juego de parámetros la utilidad en CV, train y dev (script_gbm_thr_iterador.R) con el fin de encontrar el mejor juego de parámetros que maximiza la utilidad.

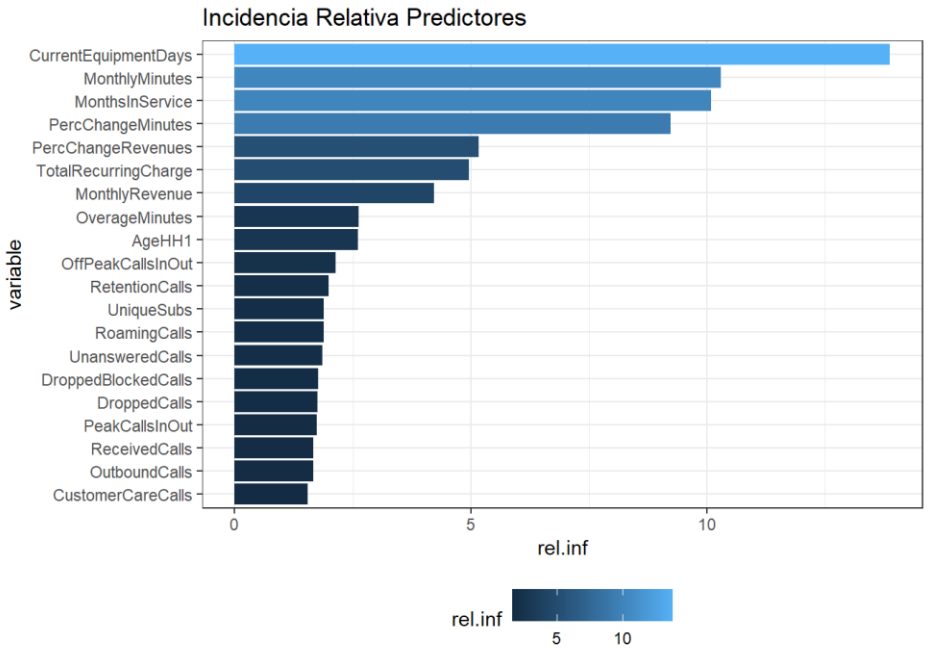
En la siguiente gráfica se muestra la evolución de la utilidad en CV, en dev y en train en función de la cantidad de árboles, para valores de shrinkage = 0.08, minobsinnode = 10 e interaction_depth = 2.



Los mejores resultados en Kaggle fueron obtenidos con los siguientes valores:

Trees=750, shrinkage=0.08, interaction.depth=2 **y** ***m.minobsinnode=10, Kaggle= 2.08205***
Trees=2500, shrinkage=0.08, interaction.depth=2 y *m.minobsinnode=30 Kaggle= 2.07658*
Trees=2750, shrinkage=0.05, interaction.depth=2 y *m.minobsinnode=10 Kaggle= 2.08479*

Algo interesante de observar del modelo de predicción que genera GBM es la incidencia relativa de cada uno de los predictores. Donde podemos observar la importancia de cada predictor, en este *dataset* siempre hay 10 predictores que siempre se encuentran en las primeras posiciones. Además estos predictores son los que se encuentran en las primeras ramas del árbol generado por RPART que se mostro en el punto 2.2.



En base a lo anterior y la incidencia de estos predictores es que se hicieron algunas pruebas utilizando solo los atributos con mayor influencia relativa, utilizando la fórmula:

Churn ~ CurrentEquipmentDays + MonthsInService + MonthlyMinutes + PercChangeMinutes +TotalRecurringCharge +PercChangeRevenues

El modelo no mejoró en cuanto a la utilidad, pero sí respecto al tiempo, en algunos casos demorando la mitad del tiempo.

2.5 OTROS

Con el fin de seguir mejorando la utilidad y por curiosidad se probaron otros algoritmos, los scripts utilizados para estos modelos se encuentran en la carpeta *"others"*:

XGBOOST. Es otra implementación de tipo *boosting*, similar a GBM. Se caracteriza por ser más rápido. Las pruebas que se realizaron nos dieron utilidades similares a GBM.

NNET. Algoritmo de redes neuronales. Es un algoritmo que consume bastantes recursos de cómputos, lleva su tiempo entrenar y las pruebas hechas no fueron buenas.

SVM. Implementación de *Support Vector Machines.* Al igual que con NNET se realizó con el fin de mejorar el resultado en Kaggle pero presentó errores durante su ejecución y se abandonó.

MODEL ENSEMBLE. En ocasiones los algoritmos se comportan mejor dependiendo de ciertas características de los datos. Puede que uno sea mejor prediciendo ciertas clases

y otro algoritmo otras. La idea del *Model Ensemble* es combinar 2 o más algoritmos para ver si el resultado final es mejor que lo que cada uno da por separado. En este caso se probó combinar *Random Forest* con *GBM*. El resultado en CV *ensemble* es mejor que cada uno de los algoritmos por separado. Se intentó validar el modelo obtenido contra *test* pero no se logró generar el archivo *.csv* con la predicción

3 Conclusiones

Respecto a CARET, es una herramienta poderosa que brinda una interfaz unificada que permite el entrenamiento de varios modelos, métricas y estrategias de validación (*cv, repeated_cv, boot*, etc). Aunque para ciertos modelos el espacio de parámetros a ajustar puede ser limitado como en el caso de RPART en el que solamente se puede ajustar el CP, y RANGER donde la cantidad de árboles está fijada en 500.

En cuanto a la selección de los parámetros de cada modelo, si bien se intentó ser sistemático en la metodología, el procedimiento fue variando de un modelo a otro en especial GBM que tiene varios parámetros para modificar.

De los modelos explorados, el que mejores resultados brinda en cuanto a utilidad en CV, DEV, train y test (Kaggle), fue GBM.

El modelo de Random Forest (Ranger) arrojó resultados aceptables, pero con un importante sobreajuste.

Los modelos de GLMNET y RPART, aunque mucho más rápidos que los dos anteriores, presentan una performance (utilidad en el entorno de 1.35) bastante inferior.

Como contrapartida, a pesar de ser mecanismos de clasificación basados en árboles, tanto GBM y Random Forest, no son buenos en cuanto a la interpretabilidad del modelo. Característica que sí es posible usando RPART, especialmente con valores de CP altos.

En resumen, el mejor modelo, el que maximiza la utilidad (2.08205) en test (Kaggle), es el obtenido con **GBM**. Con los siguientes parametros;

- n.trees= 750
- Shrinkage=0.08
- Interaction.depth=2
- m.minobsinnode=10

Aunque hay modelos de GBM de menor complejidad (menos arboles, menor *interaction depth* y mayor *shrinkage*) que devuelven buenas utilidades (entre 1.7 y 1.9) estos no fueron seleccionados. Por qué el objetivo fue buscar el modelo que retorne mayor utilidad en Kaggle y no el menos complejo.