Universidad de Buenos Aires

Facultad de Ingeniería

75.06 - Organización de Datos

Trabajo Práctico 2

Segundo cuatrimestre de 2020
Grupo: AristoDatos

Chávez Cabanillas, José E.	96467	jchavez@fi.uba.ar
Tadini, Santiago	104439	stadini@fi.uba.ar
Fedele Edo, María Cecilia	102573	mfedele@fi.uba.ar
Hojman de la Rosa, Joaquin Guido	102264	jhojman@fi.uba.ar

Índice

1. Introducción **2**
1.1. Introducción a Machine Learning y objetivos . . . 2
1.2. Repositorio . . . 4
1.3. Video . . . 4

2. Feature Engineering **5**
2.1. Problemas del set de datos (y cómo los resolvimos) . . . 5
2.1.1. Ajustando y eliminando columnas . . . 5
2.1.2. El problema del Target Leakage . . . 5
2.2. Mean Encoding . . . 6
2.2.1. Problema del Mean Encoding (y sus posibles soluciones) . . . 6
2.3. Nuevos features . . . 7
2.3.1. Antes de Mean Encoding . . . 7
2.3.2. Después de Mean Encoding . . . 9
2.4. Completación de valores nulos . . . 11

3. Modelos **12**
3.1. Random Forest . . . 12
3.2. XGBoost . . . 13
3.3. LightGBM . . . 15
3.4. KNN (k-nearest neighbors) . . . 17
3.5. Ridge . . . 18
3.6. Red Neuronal . . . 20
3.7. Ensamblando modelos . . . 21

4. Optimización de hiperparámetros **23**

5. Conclusiones **25**

1. Introducción

En esta primera sección del trabajo se comentarán algunos fundamentos introductorios y conceptos de machine learning, tema sobre el cual se desarrolla dicho trabajo. Se presentará el objetivo y se plantearán una serie de pasos para lograr el mismo. Al final de la sección se encontrará el link al repositorio online donde puede verse el código generado para el desarrollo del presente informe, y asimismo el link al video explicativo del trabajo realizado.

1.1. Introducción a Machine Learning y objetivos

El objetivo del segundo trabajo práctico de la materia Organización de Datos fue, utilizando el set de datos de ventas de la empresa Frío-Frío usado en el primer trabajo práctico, resolver un problema de Machine Learning, el cual consistió en, para cada oportunidad de negocio, estimar su probabilidad de éxito (Closed won).

Los algoritmos utilizados para la resolución del problema planteado están nucleados bajo la técnica conocida como Aprendizaje Supervizado, es decir algoritmos que consisten en crear una función capaz de predecir el valor correspondiente a cualquier objeto de entrada válida después de haber visto una serie de ejemplos, los datos de entrenamiento provistos por nosotros. Dentro del Aprendizaje Supervisado hay dos tipos de técnicas para predecir: Clasificación y Regresión.

Cuando usamos clasificación, el resultado es una clase, entre un número limitado de clases. Con clases nos referimos a categorías arbitrarias según el tipo de problema. Por ejemplo, si queremos detectar si un correo es spam o no, sólo hay 2 clases. Y el algoritmo de machine learning de clasificación, tras darle una serie de datos del correo electrónico, tiene que elegir a qué clase pertenece: spam o no-spam.

Cuando usamos regresión, el resultado es un número. Es decir, el resultado de la técnica de machine learning que estemos usando será un valor numérico, dentro de un conjunto infinito de posibles resultados. Como en este trabajo lo que queremos estimar es la probabilidad de que una venta se concrete, es un problema de Regresión. Por lo que a partir de ahora cada algoritmo que mencionemos estará siendo usado en su modo regresivo. Como aclaración, vale destacar que la mayoría de los modelos o algoritmos de machine learning permiten ambos tipos de funcionamiento, como clasificación o como regresión.

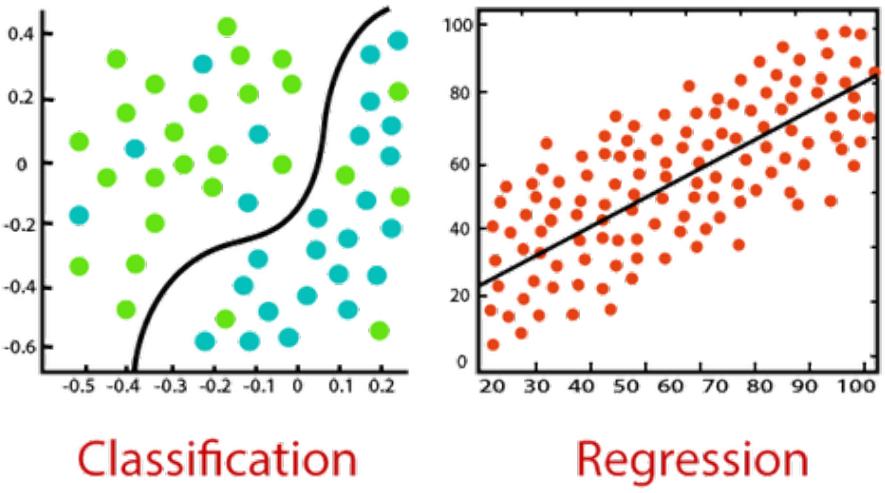

Figura 1: Clasificación Vs. Regresión

Para medir la calidad de un modelo de machine learning existen muchas métricas de error. En este caso, al ser un problema de regresión en el que calculamos probabilidades, la métrica utilizada es la ecuación de log-likelihood, en la que si tenemos un vector Y binario con el éxito real de cada oportunidad (extraído del set de datos) y un vector Y sombrero de probabilidades continuas (calculadas por nuestro modelo predictivo), la función error queda definida de la siguiente manera:

$$\text{loss}(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

Figura 2: LogLoss

Observamos que para cada elemento de la sumatoria el término que cuenta es el izquierdo si la oportunidad es exitosa y el derecho en caso contrario. La utilización del logaritmo suaviza la penalidad a la vez que tiene una derivada sencilla y facilita las cuentas. Hay un (-1) porque los logaritmos de números en el rango (0,1) son menores a cero y generalmente definimos la función objetivo como un error a minimizar. Esta función no fue implementada por nosotros ya que la mayoría de los paquetes de machine learning, entre ellos los que utilizamos, ya la incluyen.

Como se comento antes, para entrenar al modelo debe utilizarse un set de datos llamados Datos de Entrenamiento, sobre los cuales ajustaremos nuestro modelo, y luego tendremos un set llamado Datos de Test, sobre el cual haremos nuestras predicciones, y lo evaluaremos mediante la función de Logloss recién presentada.

Este trabajo se realizo en el contexto de una competencia de Kaggle, que es una plataforma muy conocida de competencias de machine learning. En ella nosotros fuimos subiendo las predicciones de nuestro modelo.

Los pasos a seguir para el diseño, configuración y uso de un pipeline completo de machine learning fueron los siguientes, explicados brevemente ahora y detallados en las próximas secciones:

- Realizar feature engineering, encontrando codificaciones que sirvan, generando nuevos atributos, analizando cuáles atributos aportan y cuáles no.
- Probar modelos predictivos que crean que se encuadran al problema. Esto puede incluir realización de ensambles, también.
- Búsqueda de hiperparámetros óptimos de alguna manera automatizada.

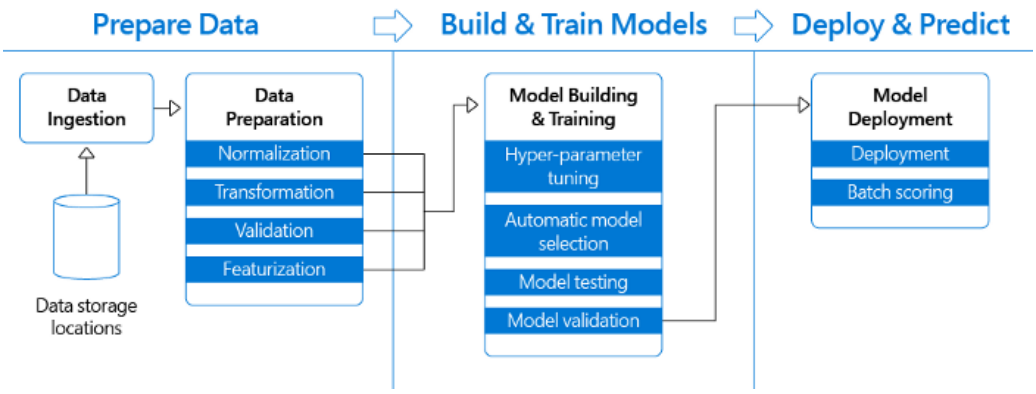

Figura 3: Pipeline de Machine Learning

1.2. Repositorio

El código generado para ayudarnos a realizar el análisis de los datos, cada paso del pipeline de machine learning y el presente informe puede encontrarse en:

`https://www.github.com/SantiagoTadini/TP2-Datos`

1.3. Video

El link al video explicativo del trabajo realizado es el siguiente:

`https://www.youtube.com/watch?v=rfLa8Zj4idM`

2. Feature Engineering

2.1. Problemas del set de datos (y cómo los resolvimos)

2.1.1. Ajustando y eliminando columnas

Lo primero que debimos realizar para procesar correctamente los datos del trabajo fue muy similar a lo desarrollado en el primer trabajo de la materia, es decir, debimos encargarnos de eliminar las columnas que no aportaban ninguna información, ya fuera por tener todos sus valores nulos, o por tener en todas las filas el mismo valor.

Observamos que, de 16947 filas en total, solo 64 (0.3776 por ciento) no son Closed Won o Closed Lost, es decir las ventas que aún estaban en proceso de negociación. Eliminamos esas filas ya que consideramos mínimo su aporte, además de que únicamente nos interesa predecir si una venta está aprobada, al no haber finalizado esas ventas, no sólo que no nos aportan ninguna información, si no que al luego binarizar nosotros el éxito (1) o fracaso (0) de la columna Stage, no nos serviría tener más valores posibles para dichas filas.

Luego otro paso importante fue, al igual que en el primer trabajo, convertir algunas columnas a otros tipos de variables que nos facilitarían el análisis y procesado. Ordenamos las filas por fecha para que no nos queden mezcladas las mismas tomando para ello como referencia la columna Opportunity_Created_Date. Y como último paso, pasamos todos los precios a Dólares. De esta manera, la relación entre los datos de esta columna empieza a tener sentido, ya que un número mayor significa que esa venta es más cara y así se logra una coherencia a la hora de comparar ventas con esta columna. Esta columna se llamo `'USD_Total_Amount'`. Luego de esto removimos del set de datos la antigua columna Total_Amount, ya que tenía valores en diferentes monedas.

2.1.2. El problema del Target Leakage

El problema más importante que debimos resolver antes de poder empezar a armar nuestro algoritmo predictor fue el problema de target leakage, “fuga de datos”. Este es conocido como uno de los problemas más difíciles al desarrollar un modelo de machine learning. Ocurre cuando entrenamos un algoritmo predictor en un conjunto de datos que incluye información que no estaría disponible en el momento de la predicción cuando se aplique ese modelo a los datos del futuro (que justamente son los que deseamos predecir). Dado que el modelo ya conoce los resultados reales, las predicciones serán irrealmente precisas para los datos de entrenamiento.

En nuestro set de datos, el problema del Target Leakage se presentaba en la columna ”Sales_Contract_No”. Esta columna indica el número de contrato... de las ventas que fueron exitosas. Es decir, para las ventas exitosas aparece un número de contrato, y para las ventas no exitosas no aparece nada. Entonces el algoritmo asume que si no se da un número de contrato en los datos a predecir, la venta será un fracaso, y si se diera un número de contrato, la venta será un éxito. El problema es que la asignación del número de contrato debería realizarse después de la venta, no antes. Podríamos decir que primero se llena la columna ”Stage”, y después la columna ”Sales_Contract_No”. En resumen, estaríamos prediciendo con información del futuro, lo que, como ya comentamos, es un problema muy grave.

2.2. Mean Encoding

Las variables categóricas presentes en los sets de datos son: *['Region', 'Territory', 'Bureaucratic_Code', 'Source', 'Billing_Country', 'Account_Name', 'Account_Owner', 'Opportunity_Owner', 'Account_Type', 'Opportunity_Type', 'Quote_Type', 'Delivery_Terms', 'Brand', 'Product_Type', 'Size', 'Product_Category_B', 'Currency', 'Last_Modified_By', 'Product_Family', 'Product_Name', 'ASP_Currency', 'Delivery_Quarter', 'Total_Amount_Currency', 'Total_Taxable_Amount_Currency']*

La forma que elegimos para transformar estas variables, para poder ser utilizadas por los modelos que no aceptan columnas categóricas, fue mediante Mean Encoding. El propósito de este método es codificar las variables categóricas en base a los labels; por ejemplo, en nuestro caso elegimos calcular el promedio del Target para cada valor categórico posible. Este sistema permite reemplazar cada columna por una sola, sin importar la cantidad de valores posibles, lo cual es una ventaja ya que no afecta el volumen de los datos y ayuda a un aprendizaje más rápido. Además, permite establecer un orden entre las variables, ya que se ordenan de acuerdo a la probabilidad de que ese valor implique que el target sea 1.

El procedimiento que realizamos fue primero agrupar según la variable categórica a codificar y calcular, para cada grupo, el promedio de "Target"; este paso se hizo sobre el set de entrenamiento, que, al haber ordenado al principio de acuerdo a la fecha de creación de la oportunidad, es lo primero que sucede en el orden temporal. Luego mapeamos estos promedios conseguidos en los distintos datasets (train, test, y el conjunto de datos para predecir y subir a Kaggle).

2.2.1. Problema del Mean Encoding (y sus posibles soluciones)

Este método de codificación de variables categóricas implica un peligro de filtración de información del label a predecir a los features de entrenamiento.

Buscando adelantarnos al problema de overfitting que este peligro puede provocar, el grupo barajó distintas posibles soluciones que fueron llevadas a la práctica para ver sus resultados. Estas soluciones se basan en penalizar la presencia del Target en la codificación, y son las presentadas a continuación:

- *Smoothing*: consiste en "suavizar" el promedio obtenido en Mean Encoding incluyendo el promedio global de todos los labels, en nuestro caso, el promedio global de la variable "Target". Este método se aplica sobre las columnas sin Mean Encoding y la fórmula es la siguiente:

$$\frac{mean(target) * nrows + globalmean * \alpha}{nrows + \alpha}$$

donde:

mean(target) es el valor del Mean Encoding tradicional

nrows es la cantidad de muestras

globalmean es el promedio global de la columna objetivo

alfa controla la cantidad de "suavización" o regularización a usar (cuánta importancia se le da al promedio global). Si alfa=0, es Mean Encoding como siempre; y si alfa tiende a un valor muy grande, la codificación va a tender al promedio global.

- *Inclusión de ruido*: este sistema lo que hace es degradar la calidad del encoding en el set de datos. Este método se aplica sobre las columnas ya codificadas con Mean Encoding.

  Para llevar esto a la práctica, utilizamos la función random.normal() de la librería numpy. Ésta extrae muestras aleatorias de una distribución normal (gaussiana). La función de densidad de probabilidad de la distribución normal a menudo se denomina curva de campana debido a su forma característica. Las distribuciones normales ocurren a menudo en la naturaleza. Por ejemplo, describe la distribución común de muestras influenciadas por un gran número de pequeñas perturbaciones aleatorias, cada una con su propia distribución única.

  A la función previamente mencionada le pasamos, inicialmente, el promedio de la columna, el desvío estándar de la misma (calculado con la función std()) y la cantidad de muestras. Cambiando el desvío estándar, se puede regular la degeneración de los datos, siendo esta menor o mayor de acuerdo al valor de desvío estándar pasado.

Aplicando estos métodos, y probando varios valores tanto para alfa en Smoothing y para el desvío estándar del Ruido, pudimos observar que no hubo mayores cambios, por lo tanto, concluimos que no hubo mayores incidencias de filtración del target a los features, y por lo tanto el modelo llegó a su máxima expresión. Además, si bien disminuyó la diferencia entre el error de train y el error de test, el error de prueba fue mayor, y como el enfoque del trabajo está en la reducción del mismo, consideramos sacarlo.

2.3. Nuevos features

2.3.1. Antes de Mean Encoding

Para la realización de nuevos features, primero se hizo un análisis de cuáles eran las columnas ya presentes en el set de datos que más influían en la toma de decisiones. Analizando la feature importance con un modelo de Random Forest, se obtuvo el siguiente gráfico.

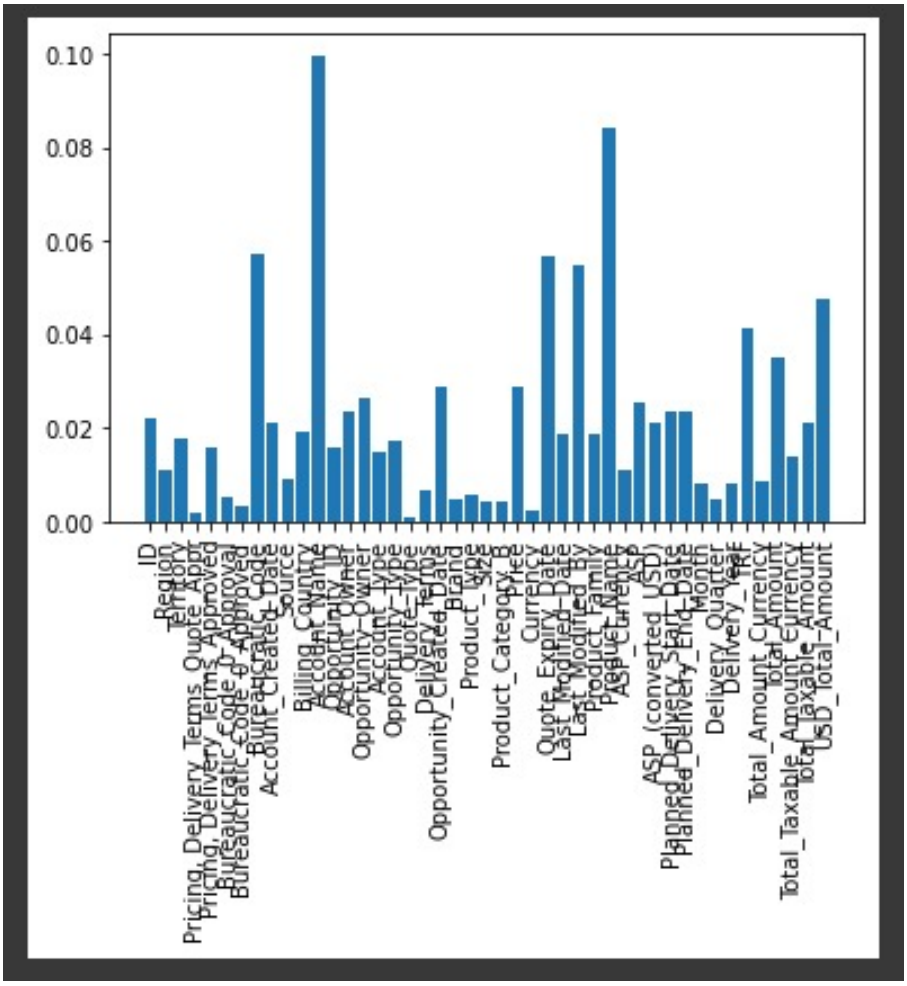

Figura 4: Feature Importance previo a nuevos features

Como podemos ver en el gráfico, hay dos columnas que aportan más información, sobrepasando a las demas por una diferencia relativamente amplia. Estas dos columnas son `'Account_Name'` y `'Product_Name'`, es decir, el nombre del cliente y el producto involucrado en la venta. A continuación aparecen `'Bureaucratic_Code'`, `'Quote'_Expiry_Date'`, `'Last_Modified_By'` y `'USD_Total_Amount'`. Guiándonos por estas columnas, se realizaron los siguientes features:

- *Cantidad de ventas por cliente por mes*: Teniendo en cuenta la importancia de los clientes en cada venta, se decidió realizar una nueva columna que indique cuantas ventas se realizaron para ese mismo cliente en ese mismo mes. Tenemos en cuenta las ventas que fueron exitosas como las que no, de manera tal que no utilizamos el Target para calcular este feature. Esto tiene la ventaja de que se puede realizar la misma operación en el set de test de manera precisa.
- *Cantidad de ventas por cliente por año*: Este feature es muy similar al anterior, solo que calcula la cantidad de ventas que se realizaron para ese cliente en ese año. Simplemente es una columna adicional con información presente en el set de datos, pero no visible en una columna.

- *Cantidad de ventas por producto por mes*: Este feature funciona de la misma manera que funciona el primer feature mencionado, pero esta vez agrupando las ventas por el producto vendido. De esta forma se genera una nueva columna con la cantidad de ventas relizadas para ese producto en ese mes.

- *Cantidad de ventas por producto por año*: Analizando el gráfico, se decidió hacer los mismos features que se hicieron para los nombres de los clientes con los diferentes productos vendidos. De esta manera, añadimos un nuevo feature que indica la cantidad de ventas de ese producto en ese mismo año.

- *Diferencia entre fecha de última modificación y de vencimiento del presupuesto*: Se observó que la fecha de vencimiento del presupuesto era una de las columnas que mayor feature importance obtuvo luego del análisis. Por lo tanto, se calculó la diferencia de días entre el vencimiento y la última modificación de la oportunidad. La operación realizada fue fecha de vencimiento menos fecha de última modificación. Esto diferencia aquellas oportunidades en las cuales se hizo una modificación justo antes del vencimiento (obteniendo como resultado un número positivo realitvamente chico), aquellas que no hicieron modificaciones (la misma fecha de vencimiento, es decir, resultado positivo muy alto) o aquellas que hicieron una modificación posterior al vencimiento del presupuesto (resultado negativo).

- *Aprobación de Código Burocrático*: Para este feature se tomaron en cuenta dos columans binarias: `'Bureaucratic_Code_0_Approval'` y `'Bureaucratic_Code_0_Approved'`. Es decir, aquellas oportunidades que necesitaban aprobación, y de esas, las que finalmente fueron aprobadas y las que no. Con este nuevo feature las oportunidades se dividieron en dos. Las oportunidades que necesitaban aprobación pero no la obtuvieron, y tanto las oportunidades que no necesitaban aprobacion como las oportunidades que sí necesitaban y fueron aprobadas. Esto se realizó con la resta entre la aprobación y la necesidad de aprobación, de manera tal que se obtuvo un -1 en aquellas oportunidades que no obtuvieron la aprobación necesitada y un 0 en las restantes oportunidades.

2.3.2. Después de Mean Encoding

Con el mismo criterio utilizado en el apartado anterior, guiándose por el gráfico generado con el feature importance de un modelo de Random Forest, se crearon diferentes features. Estos features tienen la particularidad de utilizar columnas que eran categóricas y ahora son numéricas gracias al mean encoding realizado. Operando con estas columnas, y algunas más, se crearon los siguientes features:

- *Promedio entre Account_Name y Product_Name*: Gracias al mean encoding realizado, fue posible calcular un promedio entre las dos columnas más importantes segun el grafico obtenido. De esta forma, se creó el feature `'Mean_Account_Product'`, el cual representa el promedio de éxito entre el nombre del cliente y el producto de la oportunidad.

- *Promedio entre Account_Name, Product_Name, Bureaucratic_Code y Last_Modified_By*: Teniendo en cuenta las 4 variables categóricas con mayor repercusión en la predicción de éxito de una oportunidad, se realizó un promedio entre el porcentaje de éxito de las 4. Esto quiere decir que, una vez calculado el mean encoding para cada una de ellas,

con el procedimiento mencionado, se realizó un nuevo porcentaje de éxito, producto del promedio de estas 4 variables.

- *Fecha de vencimiento de presupuesto por Account_Name*: Este feature relaciona dos columnas con gran importancia, generando así nueva información para los modelos. Se realizó el producto entre la columna `'Quote_Expiry_Date'` y `'Account_Name'` obteniendo como resultado la nueva columna `'Account_Por_Expiry'`

- *USD_Total_Amount por Account_Name*: De manera similar al feature anterior, se multiplicaron dos columnas de gran importancia, obteniendo como resultado la nueva columna `'Account_por_TotalAmount'`

- *Ventas exitosas por producto por mes (Estimación)*: De la misma forma que se contó la cantidad de ventas (exitosas o no) del producto en su respectivo mes, una vez calculado el porcentaje de éxito para cada producto, se realizó la multiplicación entre la cantidad calculada previamente y el respectivo porcentaje de cada producto. De esta forma se obtuvo una posible estimación sobre la cantidad de productos vendidos en cada mes. Por mas que no sea un cálculo exacto, la ventaja de realizar este cálculo es que se puede aplicar tanto al set de train como al set de test, ya que se utiliza el mean encoding calculado con el set de train, pero utilizado en ambos sets.

- *Ventas exitosas por producto por año (Estimación)*: De la misma forma que el feature anterior, se realizó una estimación de ventas exitosas mediante el producto de la cantidad de ventas de ese producto en ese año y el porcentaje de exito de cada producto. Así es como se obtuvo la estimación de ventas exitosas.

- *Ventas exitosas por cliente por mes (Estimación)*: De manera muy similar a los features anteriores, se realiza la multiplicación entre la cantidad de ventas totales por cliente por mes y el porcentaje de éxito de cada cliente, obteniendo una estimación de las ventas exitosas en el set de train y de test.

- *Ventas exitosas por cliente por año (Estimación)*: De la misma manera que el feature anterior, se calculó una estimación de la cantidad de ventas exitosas conseguidas por cada cliente en los respectivos años.

- *Average Selling Price por USD Total Amount*: La idea de este feature fue relacionar el ASP con el Total Amount de las oportunidades. Se optó por realizar el producto entre ellos, creando así la nueva columna `'ASP_por_USD'`.

Muchos de estos features demostraron ser de gran ayuda, tal y como se muestra en los gráficos de feature importance en la sección de modelos. A excepción de muy pocos, la gran mayoría aparece en estos gráficos, mostrando que se logró una mejoría en la calidad de los datos a utilizar.

Especialmente, se vio que los features `'ASP_por_USD'` y `'Account_por_Expiry'` fueron 2 de los 3 features más importantes a la hora de crear el árbol de decisión de un modelo XGBoost, demostrando la importancia de los features creados.

2.4. Completación de valores nulos

Otro de los temas que tuvimos en cuenta fue cómo manejar los datos nulos de los distintos sets de datos debido a que no todos los modelos implementados saben lidiar con nulos, sino que necesitan valores de entrada existentes. No se podían eliminar todos ya que perderíamos columnas importantes o demasiadas filas, incluso oportunidades enteras, que en especial en el set de datos de prueba de Kaggle no podía suceder. Por lo tanto, concluimos en que había que completar esos espacios.

En un principio, comenzamos realizando un análisis de los valores posibles para cada columna y tratando de llenar conscientemente los espacios nulos con alguno que tenga sentido. Para esto, buscamos llenarlos nulos con valores abarcadores como "Other" o incluso valores no determinados como "None" de la columna en cuestión. Pero nos encontramos con que no todas las variables tenían este tipo de valores, por lo que comenzamos a llenar con 0 (cero) aquellos lugares 'vacíos'. Otra de las opciones que pusimos en práctica fue completar con un valor distinto y no relacionado con todo el resto, por ejemplo, en una columna de valores positivos o 0, completamos con un valor negativo.

Sin embargo, estos métodos nos parecían poco sólidos y buscamos algo que permita completar con más lógica y haciendo un trabajo mecánico más eficiente.

Un enfoque para imputar valores perdidos es utilizar un modelo de imputación iterativo. El mismo se refiere a un proceso en el que cada feature se modela en función de las otros, por ejemplo, un problema de regresión en el que se predicen los valores perdidos. Es iterativo porque este proceso se repite varias veces, lo que permite calcular estimaciones cada vez mejores de los valores perdidos a medida que se estiman los valores perdidos en todos los features, o sea, los valores imputados anteriores se utilizan como parte de un modelo para predecir features posteriores.

Así fue que finalmente utilizamos la clase IterativeImputer de la librería sklearn. Esta clase implementa el modelo descrito anteriormente: estima cada feature en base a todos los demás. En concreto, es una estrategia para imputar valores perdidos modelando cada feature con nulos como una función de otros features de forma rotatoria. Observamos que sus parámetros predefinidos son los que mejor ajustaban a nuestro problema, entre los que se encuentra el estimador, que es el BayesianRidge().

3. Modelos

En esta sección se presentan los modelos que realizamos durante el desarrollo del trabajo. Estos no son los únicos modelos que usamos para predecir, si no que son los que otorgaron mejores resultados o que nos parece interesante analizar.

3.1. Random Forest

Esta fue nuestra primera aproximación al mundo de los modelos, el primero que decidimos codificar. Para ello, utilizamos el RandomForestRegressor que provee la librería sklearn (se importa desde `sklearn.ensemble`).

Random Forest es un algoritmo de bagging sobre árboles de decisión, donde cada árbol usa un subconjunto de los artibutos, tomados al azar y con repetición, un Bootstrap, del set de entrenamiento.

Comenzamos probando con los hiperparámetros por default, y luego modificando algunos de los mismos hasta ver cómo era el comportamiento del modelo. Buscamos achicar un poco la diferencia entre el error de test y el error de train, y así evitar la memorización del set de entrenamiento por parte del modelo, también llamado overfitting.

Overfitting es el efecto de sobreentrenar un algoritmo de Machine Learning haciendo que quede muy ajustado a las características de los datos de entrenamiento. El modelo recuerda gran cantidad de ejemplos en lugar de aprender a notar características en los datos sobre los que se quiere predecir. Como consecuencia del overfitting el modelo indica una tasa de éxito en las predicciones mayor a la que realmente tiene. Esto se notó especialmente en la diferencia entre el error obtenido localmente y el que obteníamos cuando subimos nuestras predicciones a Kaggle.

Los hiperparámetros trabajados en este modelo, y sus respectivas variaciones, son:

- `n_estimators`: es la cantidad de árboles a construir. Su valor default es 100, pero vimos que, reduciendo este valor, se obtiene un mejor resultado. Esto se debe a que no es muy grande el set de datos y el valor seteado en 100 provocaba sobreentrenamiento.

- `min_samples_split`: en nuestro caso es un número entero y, por lo tanto, representa el número mínimo de muestras necesarias para dividir un nodo interno. Su valor predeterminado es 2, pero pudimos observar que aumentando se mejora el resultado, ya que permite que los nodos finales contengan más datos (no sólo 2) y así se reduce la posibilidad de memorizar el set.

- `max_features`: es la cantidad de features, columnas, que se deben considerar al buscar el mejor split. Su valor por defecto es "auto", que significa que max_features=n_features, siendo n_features la cantidad total de features del conjunto de datos. Al igual que `n_estimators`, disminuir su valor conduce a un mejor resultado.

- `max_depth`: setea la cantidad máxima de niveles, es decir, la profundidad máxima de cada árbol. Su valor default es "None", lo que significa que los nodos se expanden hasta que todas las hojas sean puras o hasta que todas las hojas contengan menos de `min_samples_split` muestras. En nuestro modelo, se le estableció un límite a la profundidad, esto es, establecimos un valor a este hiperparámetro, con el mismo objetivo hasta ahora, reducir el overfitting.

- `max_leaf_nodes`: cultiva árboles estableciéndoles un máximo de nodos hoja de la primera mejor manera. Los mejores nodos se definen como una reducción relativa de la impureza. Si es su valor es el predeterminado, Ninguno, entonces tiene un número ilimitado de nodos hoja. En nuestro caso, pusimos una cota a este valor, reduciendo así el número de nodos hoja, o leaf nodes.
- `min_samples_leaf`: es la cantidad mínima de datos requeridos para ser una hoja. Sólo se considerará un punto de división a cualquier profundidad si deja al menos "min_samples_leaf" muestras de entrenamiento en cada una de las ramas izquierda y derecha. Su valor por defecto es 1, pero nosotros, al aumentar este valor, logramos obtener el efecto de suavizar el modelo. Esto aplica especialmente en regresión.

Finalmente, presentamos en la siguiente imagen uno de los estimadores del modelo Random Forest, cuya profundidad es la mayor, y que coincide con el parámetro max_depth, optimizado en 16. El objetivo no es leerse al completo, sino dar una aproxiamción al comportamiento del mismo.

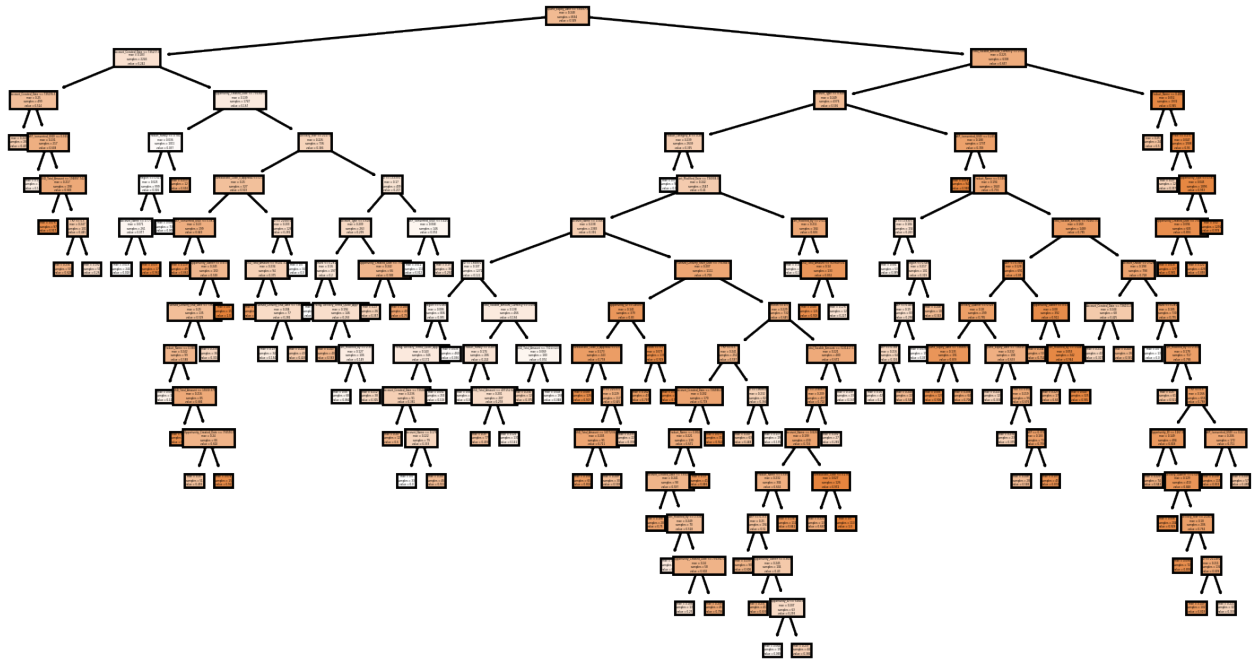

Figura 5: Uno de los estimadores de Random Forest

3.2. XGBoost

XGBoost es uno de los algoritmos de machine learning de tipo supervisado más usados en la actualidad. Es un algoritmo muy eficiente de gradient boosting en árboles de decisión.

La idea detrás del boosting es generar múltiples modelos de predicción débiles secuencialmente, y que cada uno de estos tome los resultados del modelo anterior, para generar un modelo más fuerte, con mejor poder predictivo y mayor estabilidad en sus resultados.

Este algoritmo se caracteriza por obtener buenos resultados de predicción con relativamente poco esfuerzo, en muchos casos equiparables o mejores que los devueltos por modelos más complejos computacionalmente, en particular para problemas con datos heterogéneos.

Durante buena parte del proceso de selección de modelos, XGBoost fue el que mejor resultados nos permitió obtener. Sin embargo, un problema con el que frecuentemente nos encontramos a la hora de hacer predicciones fue el overfitting, ya mencionado en el ítem anterior. La forma de reducirlo en nuestro modelo de XGBoost fue mediante la manipulación de los hiperparámetros del modelo. Estas son algunas de las medidas tomadas sobre los mismos para reducir el sobreajuste

- Aumentar el valor del hiperparámetro Gamma: este parámetro es el encargado de la regularización del modelo, o sea que aumentándolo también aumentamos la penalización del mismo. Esto genera que la complejidad del modelo disminuya, reduciendo el overfitting.
- Reducir el valor del Learning Rate: El learning rate hace que el proceso de Boosting sea más o menos conservativo, dependiendo el valor que le fijemos. Reduciendo el valor del learning rate hacemos que el árbol sea mas conservativo, con lo que el overfitting se reduce.
- Reducir Max Depth: este parámetro limita (o controla) la profundidad máxima de los árboles creados. Al reducir la profundidad reducimos la complejidad, y por ello, al igual que con Gamma, se reduce el overfitting.
- Reducir el número de estimadores: el parámetro nestimators controla la cantidad de árboles que se crean en el modelo. Al achicar este valor el modelo predecirá residuos más grandes, lo que genera que no se ajusten demasiado a los datos de train, reduciendo el overfitting.
- Reducir el valor de Min child weight: este parámetro controla la profundidad del árbol de decisión. Si lo aumentamos, disminuye la profundidad del árbol y cuando el árbol es menos profundo a su vez el modelo es menos complejo, y como ya comentamos, un modelo menos complejo es menos propenso a tener overfitting.

Por supuesto estos no fueron los únicos hiperparametros que definimos y utilizamos para el modelo. También utilizamos los siguientes: *reg lambda, reg alpha, colsample bytree, max features, subsample,eta, nthread, objective, booster, error.*

Al comienzo del trabajo fuimos definiendo a dedo los hiperparámetros, variándolos artesanalmente para mejorar el resultado y/o reducir el overfitting, pero luego los optimizamos automáticamente utilizando algoritmos dedicados a eso (más información en la siguiente sección). A continuación se detalla un análisis de feature importance para XGBoost con hiperparámetros optimizados.

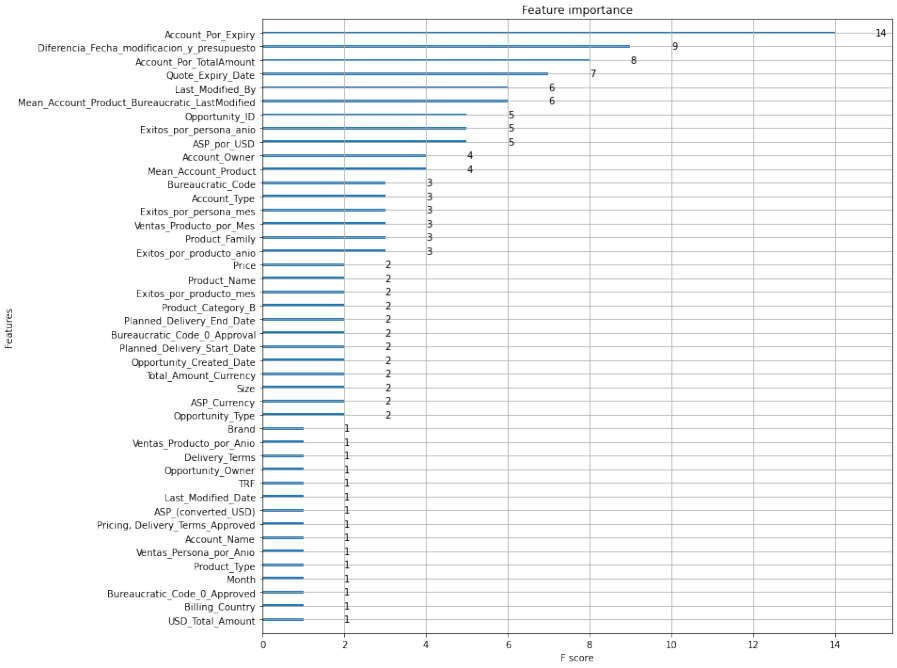

Figura 6: Feature Importance de XGBoost

Ahora veremos el árbol de decisión del mismo XGBoost:

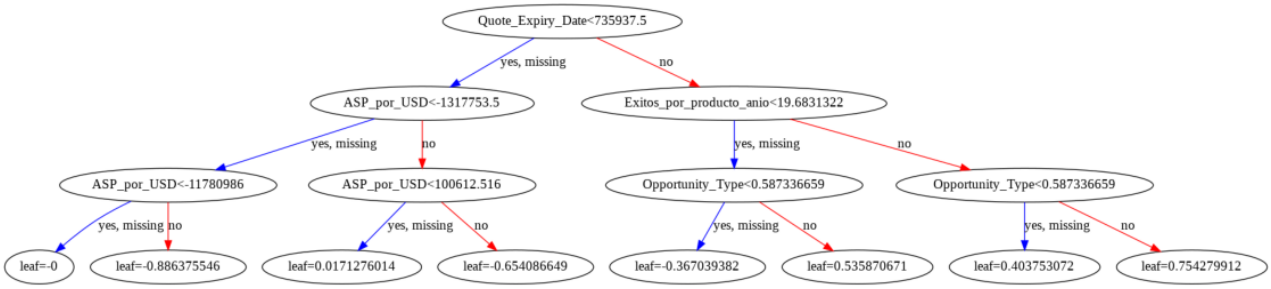

Figura 7: Árbol de Decisión de XGBoost

3.3. LightGBM

Light GBM es modelo de aprendizaje supervizado de gradient boosting que, al igual que Random Forest y XGBoost, utiliza un algoritmo de aprendizaje basado en árboles de decisión.

Una característica particular de Light GBM es que hace crecer al árbol verticalmente, mientras que otros algoritmos hacen crecer los árboles horizontalmente, lo que significa que Light GBM crece en forma de hojas de árbol mientras que otros algoritmos crecen a nivel.

No es recomendable usar LGBM en pequeños conjuntos de datos, ya que el modelo es sensible al sobreajuste y puede overfittear fácilmente datos pequeños. No hay un umbral en el número de filas, pero la experiencia general sugiere tener cuidado con los sets de datos de menos de 7.000 filas. Nuestro set de train supera esa cantidad por lo que no habría problema a priori con esto.

Junto con XGBoost, LightGBM fue el modelo de predicción que mejores resultados nos dio, pero al igual que XGBoost tuvimos bastantes problemas de overfitting, al punto de llegar

a obtener un muy buen resultado de test local, y al momento de hacer una subida a Kaggle la plataforma nos indicara un 50 por ciento más de error, lo que nos provocó unos cuantos quebraderos de cabeza.

Los hiperparámetros utilizados para la construcción del modelo de LightGBM fueron los siguientes

- Learning rate: Este parámetro determina el impacto de cada árbol en el resultado final. GBM funciona comenzando con una estimación inicial que se actualiza utilizando la salida de cada árbol. El parámetro de aprendizaje controla la magnitud de este cambio en las estimaciones.
- Num leaves: número de hojas en el árbol completo.
- Max depth: Describe la profundidad máxima del árbol. Este parámetro se utiliza para manejar el sobreajuste del modelo. Cada vez que quisimos reducir el overtfitting, intentamos primero bajar este parámetro.
- Min data in leaf: Es el número mínimo de registros que puede tener una hoja. Se utiliza para tratar el overfitting excesivo.
- Feature fraction: define la fracción de los parámetros al azar en cada iteración para construir los árboles del modelo.
- Min child weight: este parámetro controla la profundidad del árbol de decisión. Si lo aumentamos, disminuye la profundidad del árbol y cuando el árbol es menos profundo a su vez el modelo es menos complejo.

A continuación se incluye la importancia de los features para la mejor versión obtenida de Light GBM, con los hiperparámetros optimizados (más info de esto en la siguiente sección).

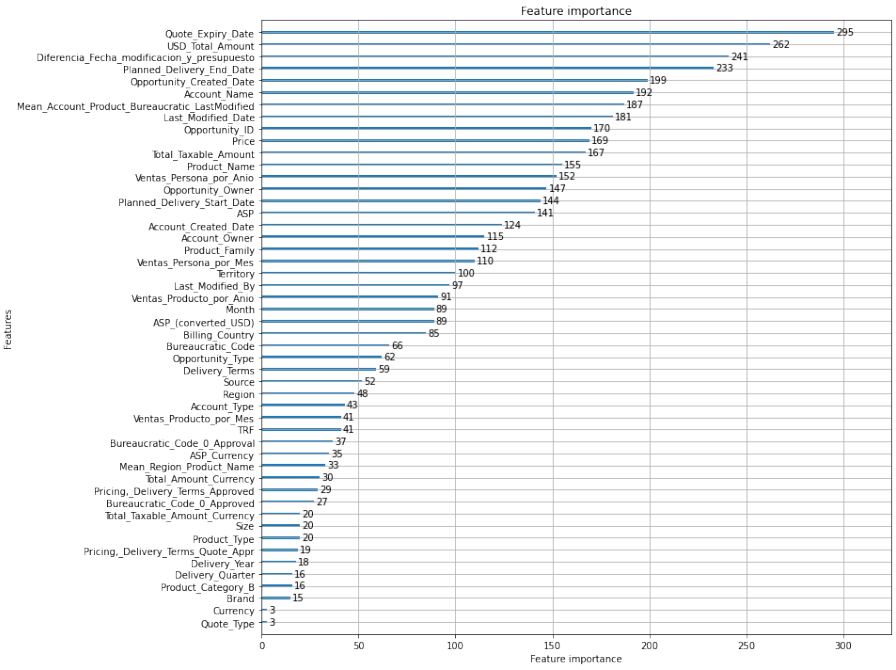

Figura 8: Feature Importance de LightGBM

Y ahora se presenta el árbol de decisión del modelo. No se espera que el lector consiga leerlo, únicamente lo mostramos para dar una idea del alcance del modelo.

Figura 9: Árboles de LightGBM

3.4. KNN (k-nearest neighbors)

Es un método de clasificación supervisada (Aprendizaje, estimación basada en un conjunto de entrenamiento y prototipos) que sirve para estimar la función de densidad $F(x/Cj)$ de las predictoras x por cada clase C_j.

Este es un método de clasificación no paramétrico, que estima el valor de la función de densidad de probabilidad o directamente la probabilidad a posteriori de que un elemento x pertenezca a la clase C_j a partir de la información proporcionada por el conjunto de prototipos. En el proceso de aprendizaje no se hace ninguna suposición acerca de la distribución de las variables predictoras.

En el reconocimiento de patrones, el algoritmo k-nn es usado como método de clasificación de objetos (elementos) basado en un entrenamiento mediante ejemplos cercanos en el espacio de los elementos. k-nn es un tipo de aprendizaje vago (lazy learning), donde la función se aproxima sólo localmente y todo el cómputo es diferido a la clasificación. La normalización de datos puede mejorar considerablemente la exactitud del algoritmo k-nn.

Este modelo en particular $KNeighborsRegressor$ dio resultados locales al principio bastante malos a los esperados, sin embargo después de establecer mejores hiperparámetros el resultado fue muy aceptable en cuanto a la predicción realizada sobre el set de datos, sin embargo al obtener un score de kaggle el error era mayor al obtenido localmente. Algunos de los hiperparámetros establecidos para esta obtención de predicción fueros los siguientes:

- n_neighbors: Número de vecinos que se utilizarán de forma predeterminada para las consultas kneighbors.
- Weights {'uniforme', 'distancia'}: función de peso utilizada en la predicción. En nuestro caso establecimos como hiperparámetro el de distancia que son puntos de peso por la inversa de su distancia. En este caso, los vecinos más cercanos de un punto de consulta tendrán una mayor influencia que los vecinos más alejados.
- Algorithm {'auto', 'ball_tree', 'kd_tree', 'brute'}:se usa kd_tree, básicamente para abordar las ineficiencias computacionales del enfoque de fuerza bruta, intentando reducir el número requerido de cálculos de distancia mediante la codificación eficiente de la información de distancia agregada para la muestra. La idea básica es que si el punto A está muy lejos del punto B, y el punto B está muy cerca del punto C, entonces sabemos que apunta A y C están muy distantes, sin tener que calcular explícitamente su distancia. De esta manera, el costo computacional de una búsqueda de vecinos más cercanos se puede reducir a

$O[DN\log(N)]$ mejor. Esta es una mejora significativa sobre la fuerza bruta para grandes N.

- Leaf_size: El tamaño de la hoja pasó a BallTree o KDTree. Esto puede afectar la velocidad de construcción y consulta, así como la memoria requerida para almacenar el árbol. El valor óptimo depende de la naturaleza del problema.
- Metric: la métrica de distancia que se utilizará para el árbol. Cabe destacar que para este parámetro en particular se usó la métrica de Manhattan, siendo la que mejor resultado dio frente a la métrica euclideana y Chebyshev.

A continuación, se presentará la comparación de los valores de hiperparámetros que contribuyeron a encontrar el error mínimo, todo esto localmente.

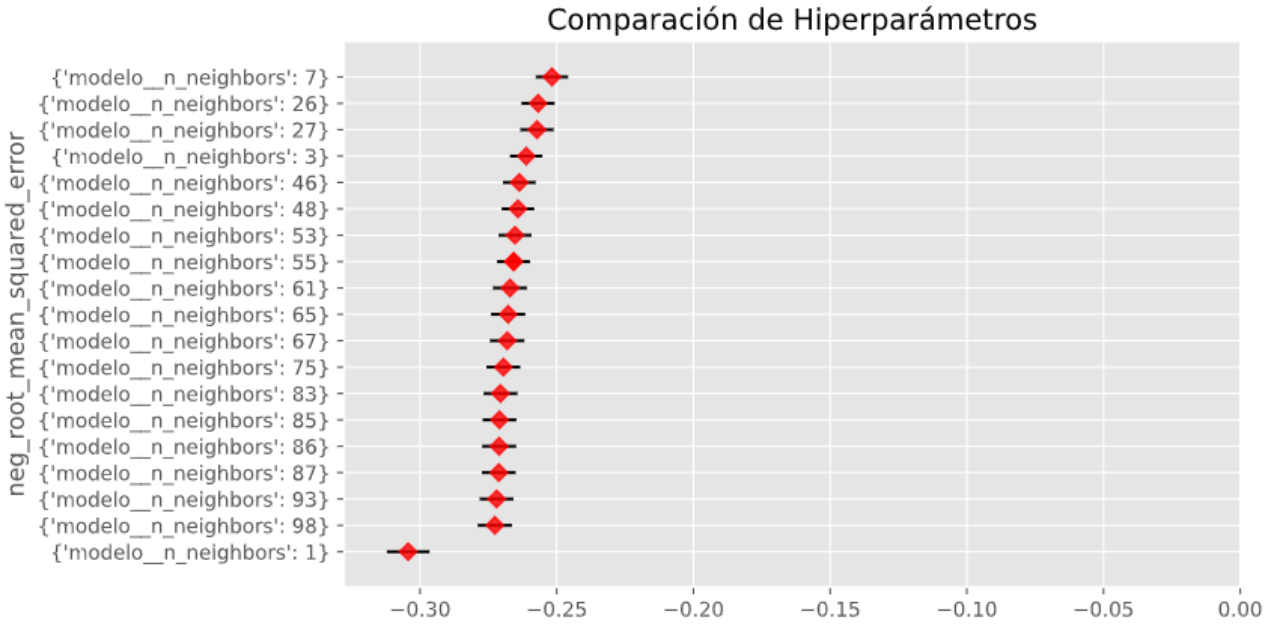

Figura 10: Comparativa de cada modelo de KNN con una cantidad de vecinos distinta

3.5. Ridge

También denominada regresión contraída o Tikhonov regularization, regulariza el modelo resultante imponiendo una penalización al tamaño de los coeficientes de la relación lineal entre las características predictivas y la variable objetivo. En este caso, los coeficientes calculados minimizan la suma de los cuadrados de los residuos penalizada al añadir el cuadrado de la norma L2 del vector formado por los coeficientes:

$$RSS_{ridge} = \sum_{i=1}^{n}(y_i - f(x))^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Donde λ es un parámetro que controla el grado de penalización: cuanto mayor éste, los coeficientes serán menores resultando más robustos a la colinealidad. Cuando α es igual a cero, Ridge es equivalente a la regresión lineal.

Es una forma de crear un modelo parsimonioso cuando el número de variables predictoras en un conjunto excede el número de observaciones, o cuando un conjunto de datos tiene multicolinealidad (correlaciones entre variables predictoras).

Produce predicciones más precisas que los modelos obtenidos por MCO + selección "clásica" de variables, a menos que el verdadero modelo sea ralo o "esparsa" (mayoría de coeficientes nulos). Sin embargo, si bien al aumentar λ (mayor penalización) los coeficientes estimados se contraen hacia cero, ninguno de ellos vale exactamente cero por lo cual no se produce selección de variables. Todas las variables originales permanecen en el modelo final.

Este modelo de Ridge no dio tantos problemas al querer calcular la predicción dado que, desde un inicio, se empezaron a buscar los hiperparámetros que mejor resultado daban y minimizaban el error en cuanto a la predicción del set de datos que se usaba. Algunos de los hiperparámetros usados para la obtención de un error local de 0.3936, fueron los siguientes:

- alpha: Fuerza de regularización; debe ser un float positivo. La regularización mejora el condicionamiento del problema y reduce la varianza de las estimaciones. Los valores más grandes especifican una regularización más fuerte.Si se pasa una matriz, se supone que las penalizaciones son específicas de los objetivos.
- fit_intercept: Ajusta la intersección para este modelo. Si se establece en falso, la intercepción no se utilizará en los cálculos (es decir, X e Y se espera que se centrado).
- normalize: Este parámetro se ignora cuando fit_interceptse establece en False. Si es verdadero, los regresores X se normalizarán antes de la regresión restando la media y dividiendo por la norma l2.
- max_iter: Número máximo de iteraciones para el solucionador de gradiente conjugado.
- solver {'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}: Solucionador para usar en las rutinas computacionales: En nuestro modelo el que mejor dio resultados fue 'cholesky' que usa la función estándar scipy.linalg.solve para obtener una solución de forma cerrada.

A continuación se muestran algunas figuras que resaltan la obtención de los mejores hiperparámetros:

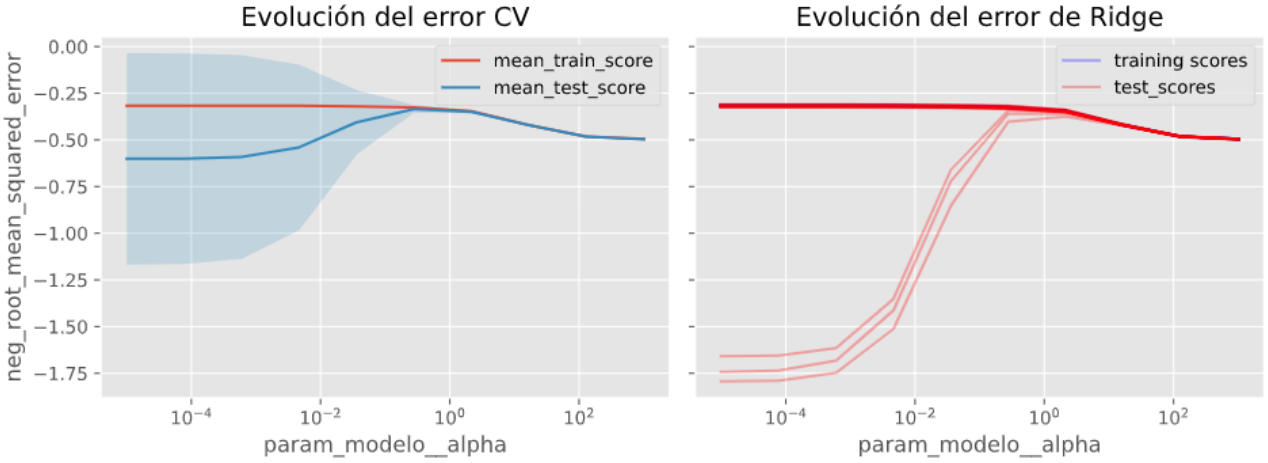

Figura 11: Error cometido con cada variación de los hiperparámetros

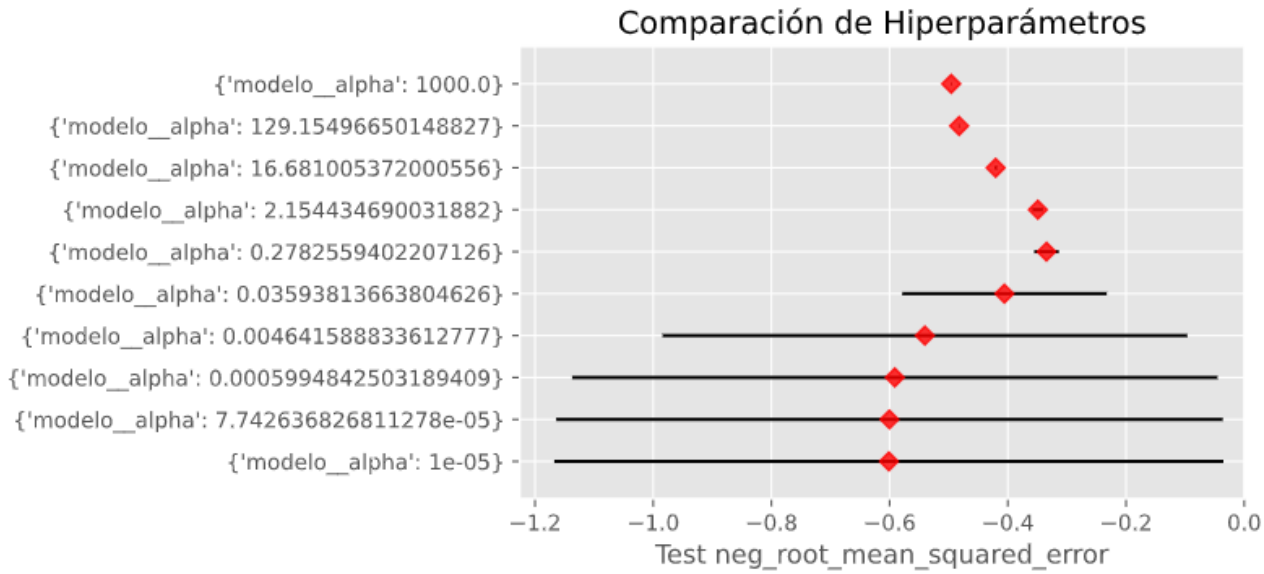

Figura 12: Comparativa de cada modelo Ridge con un alfa diferente

3.6. Red Neuronal

Las redes neuronales son modelos de predicción que intentan simular el funcionamiento del cerebro humano. Con la ayuda de "neuronas", se crea una red neuronal, compuesta de una gran cantidad de las mismas. Estas neuronas están ordenadas en capas. Una capa de entrada y una de salida. Las capas de entrada son los datos presentes y la capa de salida son los valores de las distintas predicciones.

A su vez, se pueden agregar capas entre medio de estas dos, las cuales se llaman capas ocultas. Para este trabajo, creamos dos redes neuronales distintas, una con una capa oculta y otra con dos capas ocultas.

Cada neurona tiene un peso distinto, y este peso se ajusta a la hora de entrenar. Por medio de iteraciones, al entrenar se acomodan los pesos para poder predecir de la mejor forma posible. El numero de iteraciones juega un papel clave, ya que muy pocas iteraciones significaria no encontrar los pesos optimos y muchas iteraciones podría producir overfitting. Se notó la presencia del overfitting a la hora de subir resultados a Kaggle, dando un resultado mucho peor del que se habia obtenido localmente.

Otra técnica para evitar el overfitting es utilizar dropout. Esta técnica abandona algunas neuronas dentro de la red, y de esa forma logra disminuir el sobreajuste del modelo. También se agregó un dropout despues de cada capa oculta.

Estos parámetros (cantidad de iteraciones, cuántas neuronas dejar de lado) es lo que se llaman hiperparámetros. Para que el modelo haga la mejor predicción posible, se debe encontrar los valores óptimos de estos hiperparámetros. En la busqueda de las mejores redes nueronales, se optimizaron los siguientes hiperparámetros:

- *Epochs*: Este hiperparámetro representa la cantidad de iteraciones para entrenar los pesos de cada neurona. Como se dijo previamente, se deben realizar la cantidad exacta de iteraciones para poder tener pesos óptimos sin producir overfitting. Se vio que un número alrededor de 50 producía un buen resultado.

- *Batch Size*: Es el número de datos que tiene cada iteración de un ciclo. Es útil porque permite optimizar los pesos de las neuronas sin utilizar una gran cantidad de datos y consumir memoria. El valor óptimo encontrado fue 1500.
- *Dropout*: Como se mencionó previamente, dropout refiere a la cantidad de nueronas dejadas de lado despues de cada capa oculta. El valor óptimo para una capa oculta fue 0.2 y para dos capas ocultas fue 0.06.
- *Número de neuronas*: Hubo que hacer dos análisis distintos para la cantidad de nueronas a utilizar. Uno con la red que tenía una sola capa oculta y otro para la red que tenía dos capas ocultas. El valor óptimo encontrado para la única capa oculta fue de 512 neuronas. Sin embargo, para la red con dos capas ocultas fue de 790 para la primer capa y 180 para la segunda.
- *Optimizer*: Este hiperparámetro refiere al algoritmo utilizado para optimizar los pesos de la red en cada iteración. El algoritmo más comunmente utilizado es el 'Adam', pero el 'Adamax' demostró tener mejores resultados.
- *Init*: Refiere a cómo inician los pesos de las neuronas. No es lo mismo que inicializen todas en ceros a que cada una tenga un valor específico, ya que de esta manera se puede llegar a los valores óptimos más rápidamente y evitar una alta cantidad de iteraciones. El valor que demostró mejores resultados fue la inicialización normal.

En general, no demostró malos resultados. Localmente, rondeó un log loss de 0.5 desde el principio. Sin embargo el mejor resultado obtenido en Kaggle fue de 0.55, el cual no es un mal resultado, pero no está muy cerca de los obtenidos por los árboles de decisión. El modelo que obtuvo el mejor resultado fue una red neuronal de dos capas ocultas con los hiperparámetros seteados en los valores previamente mencionados.

3.7. Ensamblando modelos

En muchas ocasiones los mejores algoritmos de Machine Learning son combinaciones de varios algoritmos. De hecho, el método de boosting es un tipo de ensamble que consiste en, partiendo de algoritmos muy simples, construir un algoritmo muy preciso. Simplemente, se empieza de un algoritmo simple (como un árbol), se entrena, se analizan sus resultados, y luego se entrena el siguiente algoritmo simple, dándole mayor peso a los que resultados que tuvieron peor performance previamente. En el trabajo, este método fue usado en algoritmos como LightGBM o XGBoost.

Otra forma de ensamble es el bagging, que consiste en aplicar un clasificador n veces y luego promediar el resultado final. Es importante que cada una de las corridas del clasificador sea con datos reemplazados en el set de entrenamiento (bootstrapping). La principal ventaja generada es evitar el overfitting, ya que ningún clasificador conoce el set de entrenamiento completo y por lo tanto no se puede sobreajustar al set de datos. Random Forest utiliza esta técnica para entrenar.

El método de ensamble que nosotros utilizamos en nuestro trabajo fue el método conocido como Averaging, que consiste en quedarse con el promedio de las predicciones de los modelos a ensamblar. Es decir, de todos los algoritmos corridos, se hace averaging de combinaciones de estos. Es un método muy útil para reducir el overfitting.

En nuestro trabajo nos encontramos con la siguiente situación: nuestros mejores modelos eran dos: un modelo XGBoost con un error relativamente bajo, pero que overfitteaba muy poco. Este error era de un 0.41 local aproximadamente y subiendo a Kaggle nos daba aproximadamente 0.46. Luego teníamos un modelo LightGBM que daba un error más bajo localmente que XGBoost, que era de 0.39 aproximadamente. Sin embargo, al momento de subir a Kaggle veíamos un overfitting mayor, llegando a un resultado en la plataforma de 0.47.

Lo que hicimos con ambos modelos fue ensamblarlos mediante la técnica de Averaging, dándole 50 por ciento de peso a cada uno. Cabe destacar que podríamos haberle dado más de la mitad del peso a alguno de los modelos, es más, si tuviéramos más modelos podríamos repartir ese peso de la forma que quisiéramos, es decir que podríamos no habernos limitado a solo dos modelos. Llegamos a probar una tríada con XGBoost, LightGBM y Random Forest, pero sin llegar a muy buenos resultados.

Entonces luego de ensamblar XGBoost y LightGBM obtuvimos nuestro mejor puntaje en Kaggle, que es de aproximadamente 0.453.

Algo que cabe destacar de Averaging es que este método da mejores predicciones cuando los resultados no tienen correlación entre sí. Es decir que tendría más sentido ensamblar un árbol de decisión con una red neuronal, ya que no siguen la misma lógica interna y los resultados mejoran. Sin embargo, como los únicos modelos que dieron resultados relativamente bajos fueron árboles, decidimos ensamblar sólo entre ellos para el resultado final.

Otro tipo de ensamble que probamos entre XGBoost y LightGBM, siguiendo un poco la idea de Averaging, fue quedarnos con la probabilidad más alta de entre las posibles. Es decir, para cada fila nos quedamos con el valor máximo de probabilidad entre los dos posibles. Esta forma de armar el set final de predicciones dio un error ligeramente mayor al obtenido utilizando Averaging, pero sin dudas más que aceptable. Siguiendo esta línea también probamos quedándonos con la mínima probabilidad, obteniendo un error ligeramente mayor al máximo.

El ensamblado es el arte de combinar un conjunto de diversos de modelos con el fin de lograr la mejor predicción posible con el mínimo error. Sin duda el ensamble de modelos es una herramienta muy poderosa, y bien usada puede llegar a mejorar drásticamente los resultados de una predicción. Las grandes competencias de machine learning, entre ellas las de Kaggle, prácticamente siempre son ganadas por equipos que utilizan ensambles de decenas o incluso cientos de modelos diferentes, lo que verdaderamente da una idea del poder del ensamble.

4. Optimización de hiperparámetros

A lo largo de todo el trabajo práctico hicimos uso de tres métodos de optimización distintos:

1. **Random-Search**: utilizamos la implementación correspondiente a la librería sklearn, RandomizedSearchCV(). Esta fue nuestra primera aproximación a la optimización de hiperparámetros.

   Esta función implementa la búsqueda aleatoria de hiperparámetros óptimos. Para ello se establece un espacio de parámetros a probar, pero no se prueban todos ellos, sino una cantidad limitada, n_iter, que por default es 10. Nosotros incrementamos este número para que pueda realizar más pruebas.

2. **Grid-Search**: Un enfoque básico y sencillo para optimizar los hiperparámetros es la técnica Grid Search (GS). Básicamente, se define y evalúa una lista de valores candidatos para cada hiperparámetro. El nombre grid viene del hecho de que todos los posibles candidatos dentro de todos los hiperparámetros necesarios se combinan en una especie de cuadrícula. A continuación, se selecciona la combinación que produce el mejor rendimiento, evaluándola en un conjunto de validación.

   Esta forma de optimización no dio buenos resultados en nuestros modelos basados en árboles. Entendemos que es un método que sirve mejor para problemas de clasificación, donde los resultados se clasifican en alguna categoría, que para problemas de regresión donde los resultados son valores continuos.

   Con pequeños conjuntos de datos y muchos recursos, Grid Search producirá resultados precisos. Sin embargo, con grandes conjuntos de datos, como el caso de nuestros sets, las grandes dimensiones ralentizarán enormemente el tiempo de cálculo y resultarán muy costosas.

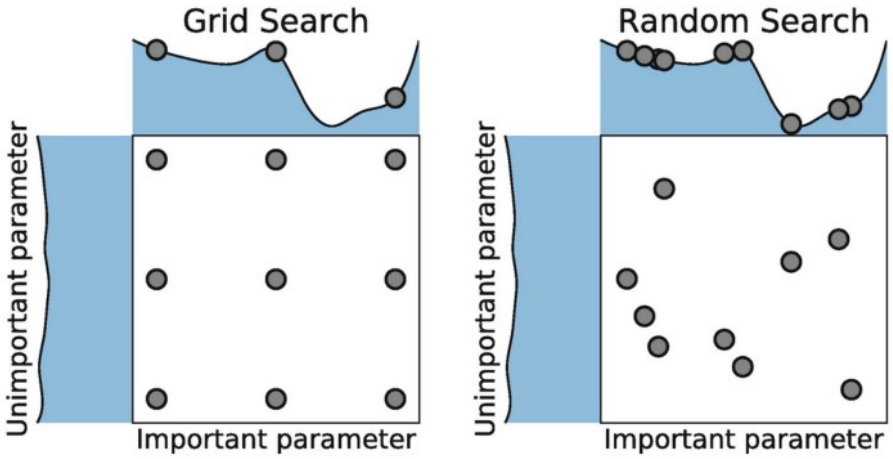

Figura 13: Grid Search vs Random Search

3. **Búsqueda Bayesiana**: en búsqueda de un optimizador más robusto, hicimos uso de la librería HyperOpt en el trabajo. En la misma, actualmente, se implementan tres algoritmos: Random Search, Tree of Parzen Estimators (TPE) y Adaptive TPE.

   En concreto, para implementar este optimizador, comenzamos estableciendo la función objetivo, la cual recibe un diccionario que contiene los hiperparámetros a probar en esa iteración. Dentro de la misma, generamos un modelo seteado con los hiperparámetros recibidos y calcula el error, en nuestro caso, el log_loss, en cada iteración. Luego, constituimos el espacio de hiperparámetros, con los rangos dentro de los cuales probarlos, y finalmente, llamamos a la función fmin(). Esta última es la principal, cuyo propósito es minimizar la función objetivo, y a la cual le pasamos por parámetro:

   la mencionada función objetivo a minimizar

   el espacio en el cual buscar

   el algoritmo de búsqueda a usar

   la cantidad máxima de iteraciones

   El resultado de la misma son los hiperparámetros óptimos encontrados en las iteraciones realizadas, dentro del espacio dado.

   Este método de optimización fue adaptado a los distintos modelos hechos en este trabajo, y al ser la manera más sólida, fue la que dio mejores resultados.

   Además de obtener los resultados con menor error de test, algo muy importante de HyperOpt es la velocidad con la que se ejecuta, haciendo cientas e incluso miles de iteraciones en apenas unos minutos. Esta cualidad resalta frente a Grid y Random search que pueden estar muchas horas ejecutándose sin llegar a un resultado óptimo.

   Los mejores resultados obtenidos por nuestros modelos, incluyendo los mejores puntajes que conseguimos en kaggle se deben a la utilización de este método de optimización. Al principio definimos un espacio de hiperparámetros posibles muy amplio, para luego volver a correr el método acotando cada vez más este rango, hasta llegar a un valor que creemos que ya no se puede mejorar o que las mejoras son tan pequeñas que no tiene sentido seguir. Esta forma de optimización no se podría haber hecho con los otros métodos presentados ya que, como se explicó en el párrafo anterior, hubiera sido imposible por la larga duración que tienen en su ejecución.

5. Conclusiones

El objetivo del segundo trabajo práctico de la materia Organización de Datos fue el desarrollo de un pipeline completo de machine learning, es decir la realización del proceso de feature engineering, la búsqueda y prueba de modelos predictivos que encuadraran con el problema y el ensamble de estos, y luego la búsqueda de hiperparámetros óptimos para mejorar los resultados de los modelos.

Durante el desarrollo del presente trabajo podríamos decir que atravesamos momentos de felicidad, tristeza, e incluso ira. La enseñanza de contenidos teóricos brindados en la materia nos permitió abordar el inicio del trabajo con una base de conocimientos muy amplios, pero con poca idea práctica de cómo comenzar. Es por eso que debimos investigar mucho sobre paquetes, modelos, y maneras de iniciar el pipeline que queríamos desarrollar.

Consideramos que el proceso de feature engineering fue el que nos dio mas quebraderos de cabeza. Si bien resolvimos rápidamente el problema de target leakage del tp, tuvimos otras complicaciones, como la dificultad inicial de pensar qué features nuevos implementar, y luego el testeo de si realmente estaban aportando o no a la predicción de los modelos. Finalmente terminamos decidiendo en base al Feature Importance de algún modelo, es decir si implementábamos un feature nuevo, este cobraba mucha importancia en el modelo, pero el resultado empeoraba mucho, el feature era descartado. Si al contario el resultado mejoraba, se consideró que el feature estaba ayudando al modelo, y se procedió a dejarlo.

Un error inicial con este último proceso fue la eliminación de columnas por pensar que no aportaban nada, para luego comprender que para los modelos de machine learning, cuanta más información haya, mejor.

Sin embargo, el problema que más nos hizo sufrir, y que nos molestó desde el primer día hasta el día de la entrega, fue el problema del overfitting. Incontables veces creímos haber alcanzado un muy bajo error, para luego subir las predicciones a Kaggle y casi llorar de desesperación al ver que el resultado había empeorado de manera drástica, a veces incluso duplicando el valor de error local. La forma que tuvimos de ir paliando este problema fue trabajar con los hiperparámetros que descubrimos que podían ayudarnos a reducir el overfitting. Además, el ensamble de modelos también colaboró con esta ardua misión.

Una vez superados los mayores problemas, consideramos que logramos armar modelos predictivos robustos y confiables, disminuyendo todo lo posible la brecha entre el error local y el indicado por la plataforma. Si bien al principio fuimos poniendo los hiperparámetros medio a dedo, luego haciendo uso de los diferentes algoritmos y métodos de optimización, sobre todo los de la libería HyperOpt, logramos obtener valores para los hiperparámetros que redujeron significativamente el error y lograron mucha más precisión en las predicciones realizadas.

La necesidad de investigación propia, tanto para saber cómo armar los features, como para descubrir y utilizar nuevos modelos y métodos de investigación fomentó una profundidad en la búsqueda de los mismos que pocas veces habíamos realizado antes, tanto en esta materia como en otras que hemos cursado. Consideramos muy valioso este proceso ya que es muy común en la industria que nuclea nuestras carreras la necesidad de investigación propia sobre métodos y modelos desconocidos en casi cualquier tópico.

Ademas, pensamos que el hecho de que el trabajo se realizara bajo una competencia de Kaggle fue provechoso para nosotros, ya que nos dio un motivo firme para tratar de mejorar aún más nuestros modelos en todo momento. Además, de no ser por la competencia no hubiésemos tenido manera de comprobar dos cosas: la primera es cuán bien funcionaba nuestro modelo,

ya comentamos antes que nos ocurrió de lograr un error muy bajo localmente, para que luego la plataforma nos indicara que ese error era mucho mayor. Fue duro pero provechoso chocar contra la pared de kaggle para detectar overfitting que no sabíamos que teníamos. Lo segundo que la competencia nos permitió comprobar es si el error que estábamos obteniendo estaba bien situado, es decir al ver los errores del resto de los grupos podíamos compararnos contra ellos y saber si nuestro error era bueno o malo relativamente.

Como comentario final, comentamos que todo el grupo disfrutó la realización del trabajo. Si bien como se comentó antes hubo momentos más felices y momentos más tristes, a todos nos gustó este primer roce con el Machine Learning, una rama de la informática cada vez más importante y solicitada, y quedamos conformes con el resultado final, tanto del trabajo como del presente informe.

Por último, de parte de todo el equipo de desarrollo de AristoDatos, agradecemos profundamente la atención brindada durante la lectura del presente informe.

FIN