.UBAfiuba
FACULTAD DE INGENIERÍA

UBA - Facultad de Ingeniería

Departamento de Computación

Organización de Datos (75.06)

Trabajo Práctico 2

2do cuatrimestre - 2020

GRUPO: **Que Buena Data Papá**		
Alumno	**Padrón**	**Mail**
Xifro, Juan Bautista	101717	jxifro@fi.uba.ar
Rojas, Mateo	104985	marojas@fi.uba.ar
Re, Gabriel	105095	gre@fi.uba.ar
Kovnat, Thiago	104429	tkovnat@fi.uba.ar
Locatelli, Santiago	104107	slocatelli@fi.uba.ar

Índice

1. Introducción

El objetivo de este segundo trabajo es predecir, para cada oportunidad de negocio brindada, la probabilidad de éxito.

Para esto utilizamos **Machine Learning** con el fin de lograr lo propuesto. En el trabajo se prueban distintos algoritmos, los cuales todos en distinta manera hacen uso de los datos. Es por esto que es muy importante saber qué datos usar, y buscar cómo codificarlos de tal forma que se aprovechen de la mejor manera posible.

Los datasets utilizados fueron brindados por la cátedra, https://www.kaggle.com/c/friofrio/data. Estos sets de datos contienen oportunidades realizadas en distintas partes del mundo, en distintos años, etc.

Durante la realización del trabajo se probaron distintos algoritmos estudiados en la materia (XGBoost, Random Forest, Redes Neuronales, entre otros) con distintos features que se extrajeron del dataset dicho anteriormente. Aquellos modelos que no presentaban mejora fueron descartados y nos concentramos en mejorar aquellos que si.

2. Dataset

Primero y principal, lo que hicimos fue analizar como están conformados nuestros datos. De esta forma tenemos un conocimiento general sobre lo que vamos a trabajar. El dataset a analizar pesa **662.1 KB** . Cuenta con **16947 Filas** y **52 Columnas**.

Cada fila representa una oportunidad, cada oportunidad consiste en un proyecto de venta o instalación de equipos para un cliente. Y cada columna tiene distinta información sobre el data set, las cuales son:

Se utilizó el dataset train para entrenar y testear los distintos algoritmos utilizados durante el trabajo. Dicho dataset fue modificado a lo largo del tiempo, quitando aquellos datos que no fueran útiles; para lograr eso hemos recurrido al **Feature Engineering** y **Encoding**. A continuación detallaremos como fueron utilizados.

2.1. Feature Engineering

Sirve para *limpiar* el dataset, ayuda mucho a saber cuales columnas (features) aportan mejor información que otras y si hay que eliminarlas o crear nuevas para la optimización del algoritmo de predicción.

2.1.1. Time Series

Este feature lo utilizamos pensando al dataset como una serie temporal. Basándonos en el **Delivery Year** y **Delivery Quarter** sacamos el promedio del **Total Amount** de cada **Opportunity Owner** según el Año y Quarter y lo introducimos al dataset como **Promedio Owner Por Year And Region**. Luego agregamos el feature **Lag 1** siendo éste el promedio del **Total Amount** por cada **Opportunity Owner** del **Delivery Quarter** anterior. Conociendo ambos features podemos crear otro llamado **Delta** el cual es la resta del **Promedio Owner Por Year And Region** menos **Lag 1**. De esta forma estamos ordenamos los datos según el tiempo y creamos interacciones entre las filas para indicarle al algoritmo que éstas interaccionan.

Sabemos que hacer esto no es gratis, ya que hay que ordenar el dataset y esto puede llevar tiempo. Lo aplicamos sabiendo que la cantidad de datos no es tan grande y ordenarlo no propone mucha complicación. También afecta la forma de hacer el **Split** al dataset, hay que hacerlo de forma tal que se entrene con el pasado y se testee con el futuro. Esto es de suma importancia, ya que no se debe entrenar con el futuro.

2.1.2. Feature Transformation

La reestructuración de features o *Feature Transformation* cumple un papel importante dentro del **Feature Engineering**. No solo crear nuevas columnas es importante, sino modificar las ya existentes para *emprolijar* los datos de la misma. Cambios como aplicar el logaritmos a columnas numéricas con extremos muy separados, como los son **Total Amount**, **Lag 1** o **Delta** sirve para achicar las diferencia entre los extremos e indicarle al algoritmo de Machine Learning que los datos de esa columna siguen, en este caso, un comportamiento logarítmico.
Cabe aclarar que también se pudo aplicar la raíz cuadrada. Para decidirnos entre estas opciones, visualizamos la distribución de los datos según ambas aplicaciones.

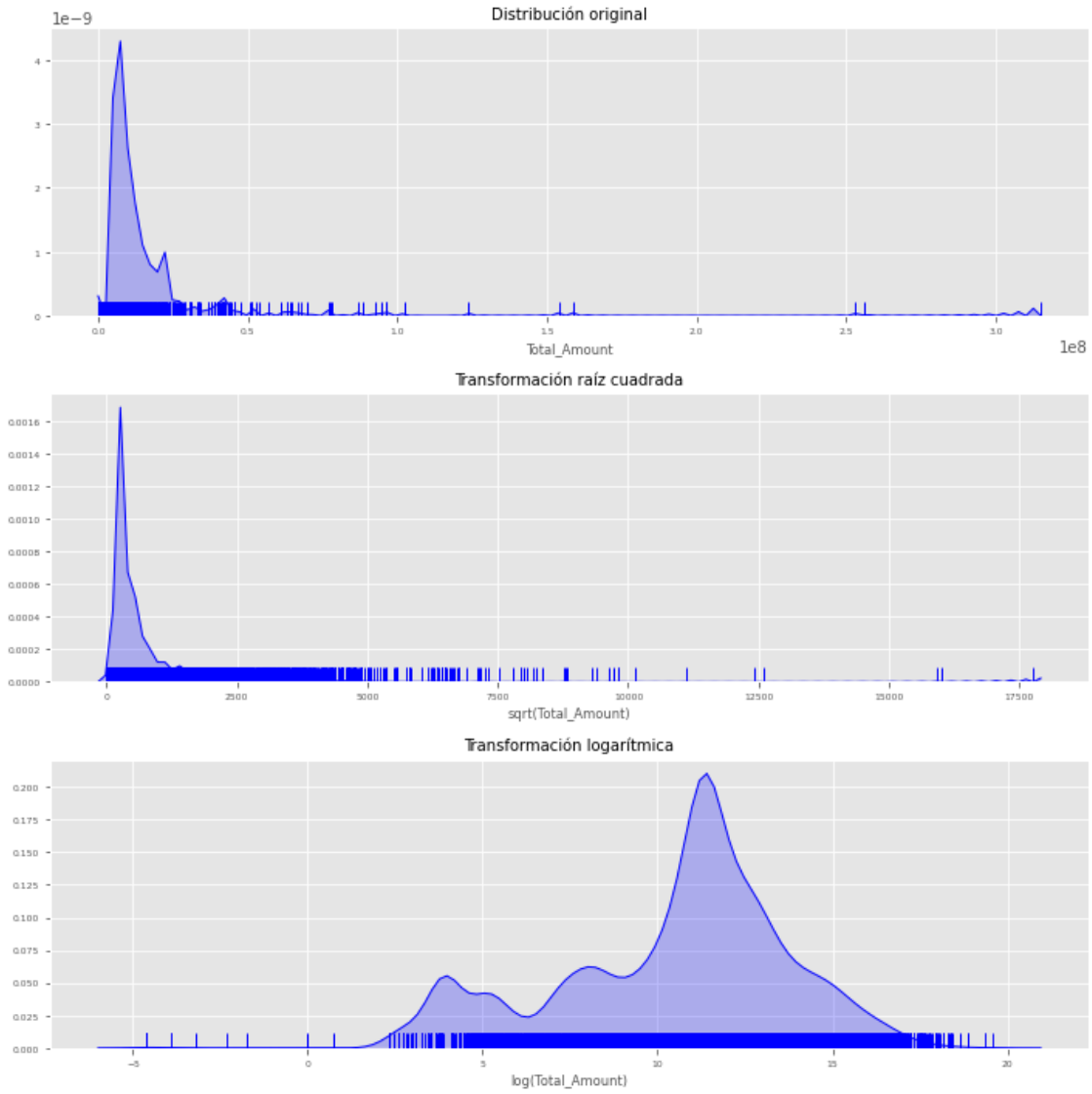

Figura 1: Análisis de distribución numérica.

Como vemos en el gráfico, tanto la distribución original como la que ofrece la raíz cuadrada es poca, mientras que la logarítmica lo hace mas equitativamente.

Otra transformación hecha fue en las columnas categóricas con muchas variables como lo son **Opportunity Owner** o **Product Name**. Lo que se hizo en este caso fue agarrar las variables que se repetían menos de 1 % y reemplazarlas por "Other" de esta forma reemplazamos los valores outliers y podemos modificar en el set de test los valores que no existen dentro del set de entrenamiento a esta misma variable.

2.2. Encoding

Se utilizó para encodear las columnas de tipo categórico y transformarlas a numérico. Este es un proceso que debe ser hecho con distintos algoritmos de encoding, ya que si se hacen cambiando cualquier variable categórica a números, el algoritmo de Machine Learning puede encontrar patrones en esos números que son completamente inventados y sin ningún tipo de sentido, perjudicando de esta forma, al performance del algoritmo.

2.2.1. OneHotEncoding

El primer OneHotEncoder que probamos para las features categóricas, importado de la biblioteca SKlearn, es bastante limitado, ya que también encodea las features numéricas, lo cual no nos servía. Es por eso que no duro mucho entre nosotros.

Teniendo en cuenta lo antes mencionado, empezamos a utilizar otra funcionalidad de la biblioteca SKlearn, el ColumnTransformer, para complementar el hecho de que el OneHotEncoder no separa las features categóricas y las numéricas.

Esta combinación de herramientas nos dio una seguridad mucho mayor con respecto a la features y encoding que estábamos utilizando.

2.2.2. Mean Encoding

La idea de este es codificar las variables categóricas en a base a los promedio de las apariciones de los labels. Sin embargo, a pesar de que esta codificación puede resultar muy útil a veces, hay que tener bastante cuidado, ya que podría llegar a entorpecer fácilmente al modelo ocasionando que aprende correlaciones inexistentes.

No utilizamos ninguna biblioteca para llevar a cabo este encoding, lo realizamos de la siguiente manera:

```
# Donde df es el Data set original, y short_df es el Data set ya pre procesado, y listo para entrenar.
# Se toma como criterio que cualquier categoria que aparezca menos de 5 veces se considere como "Other".

df["Feature"] = np.where(df.groupby('Feature')["Feature"].transform(len) > 5, df["Feature"], "Other")
mean_encoded_account = df.groupby("Feature")["Target"].mean().to_dict()
short_df["Feature"] = df["Feature"].map(mean_encoded_account)
```

2.2.3. Label Encoding

Al igual que el Mean Encoding, el Label Enconding puede ocasionar que el modelo aprenda relaciones entre features que no son ciertas, por lo que termina prediciendo mal. Pero, nos termino sirviendo para algunos casos en especifico, cuando el Mean Encoding no funcionaba como esperábamos, paso que ocurrió aquello que mencionamos como peligro de este mismo.

No utilizamos ninguna biblioteca para llevar a cabo este encoding, lo realizamos de la siguiente manera:

```
# Donde df es el Data set original, y short_df es el Data set ya pre procesado, y listo para entrenar.
# Se toma como criterio que cualquier categoria que aparezca menos de 5 veces se considere como "Other".

dict_product_name = {k: i for i, k in enumerate(df['Product_Name'].unique())} # Label Encoding
short_df["Product_Name"] = df["Product_Name"].map(dict_product_name)
```

3. Modelos

Fuimos pasando por varios modelos tanto Arboles de Decisión, como Gradient Boosting, KNN, Redes Neuronales y Ensambles. Tanto clasificadores como modelos de regresión. Gracias a esto, pudimos corroborar las fortalezas y debilidades de estos.

3.1. Arboles de Decisión

3.1.1. ID3

El primer árbol de decisión que se uso fue el ID3, utilizando features únicamente categóricos dada la restricción del modelo. Se utilizó la librería ChefBoost que nos provee una interfaz simple para crear arboles de decisión. Sin embargo, como era de esperarse dado nuestro análisis en el TP1, los features categóricos no fueron suficiente como para tener una predicción certera del resultado, dado que, como dijimos en el trabajo práctico anterior, los features mas importantes son, en su mayoría, numéricos.

3.1.2. C 4.5

También se probó con el árbol de decisión C 4.5 que presenta algunas mejores con respecto al ID3, entre ellas están:

- Aceptar features numéricos.
- Aceptar datos con atributos faltantes, se ignoran en el calculo de la entropía.
- Permitir que los atributos tengan un cierto peso.
- Poder podar el árbol una vez creado, es decir, remover las ramas que no ayudan en nada reemplazándolos con nodos hoja.

Pero aun así, este no tuvo mucho éxito, ya que nos encontramos con modelos con más capacidad que este.

3.1.3. Random Forest

El Random Forest es un modelo que construye múltiples arboles de decisión y toma como decisión el valor que más se repite entre sus arboles en el caso de un modelo clasificador y el promedio de sus arboles en caso de utilizar un modelo regressor. Para ejemplicar el funcionamiento de un árbol de decisión observemos el siguiente gráfico que muestra como funciona un árbol de decisión clasificador.

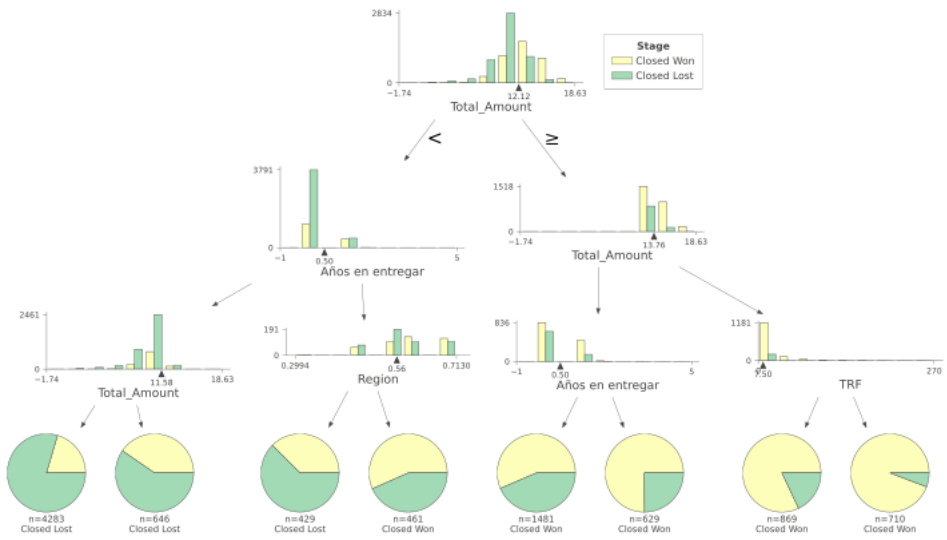

Figura 2: Árbol de Decisión Clasificador.

Podemos observar como el árbol impone una condición sobre un feature (Por ejemplo, Total Amount <12.12 para el nodo raíz) y basado en el resultado de esa condición se sigue una de las ramas que a su vez tienen sus condiciones propias. Esto se repite hasta llegar a un nodo hoja que es el que tiene la decisión final que se debe tomar. Estos arboles pueden tener mucha más profundidad, pero se utilizo uno relativamente chico por motivos de visualización, pero la lógica se mantiene para un árbol de cualquier profundidad.

Classification

Inicialmente, entrenamos un modelo clasificador utilizando features tanto categóricos como numéricos, lo que aumentó su performance en comparación con arboles de decisión más simples como ID3 o C 4.5. Los resultados, igualmente, no eran lo suficientemente acertados, ya que la precisión variaba entre un 70 y 80 por ciento. Pudimos aumentar esta precisión utilizando un Grid Search incluido en la librería SKlearn que nos ayudó a la hora de corregir los hiperparámetros del Random Forest. Sin embargo, los resultados que obteníamos en Kaggle no eran lo que esperábamos teniendo en cuenta la precisión que obtuvimos con nuestro set de test interno.

Regressor

Tras investigar las diferencias entre un modelo clasificador y uno regressor, nos dimos cuenta que podíamos transformar nuestro valor binario de resultado en un valor continuo que representaba la probabilidad de venta de una oportunidad. Otra vez utilizamos la librería SKlearn que nos provee un Random Forest Regressor que, como dijimos anteriormente, toma el promedio del resultado de sus arboles como valor final para la predicción. Desde un principio, el Regressor decremento muy fuertemente el log loss que obteníamos con nuestro set de test interno, por lo que nos decidimos orientar hacia los modelos regresores ya que se ajustaban mejor a nuestros datos. Luego de utilizar

nuevamente un Grid Search para modificar los hiperparámetros, encontramos los valores que mejor se ajustaron a nuestro set y nos dio el mejor resultado en ese momento con el set de test de Kaggle: 0.55 de Log Loss.

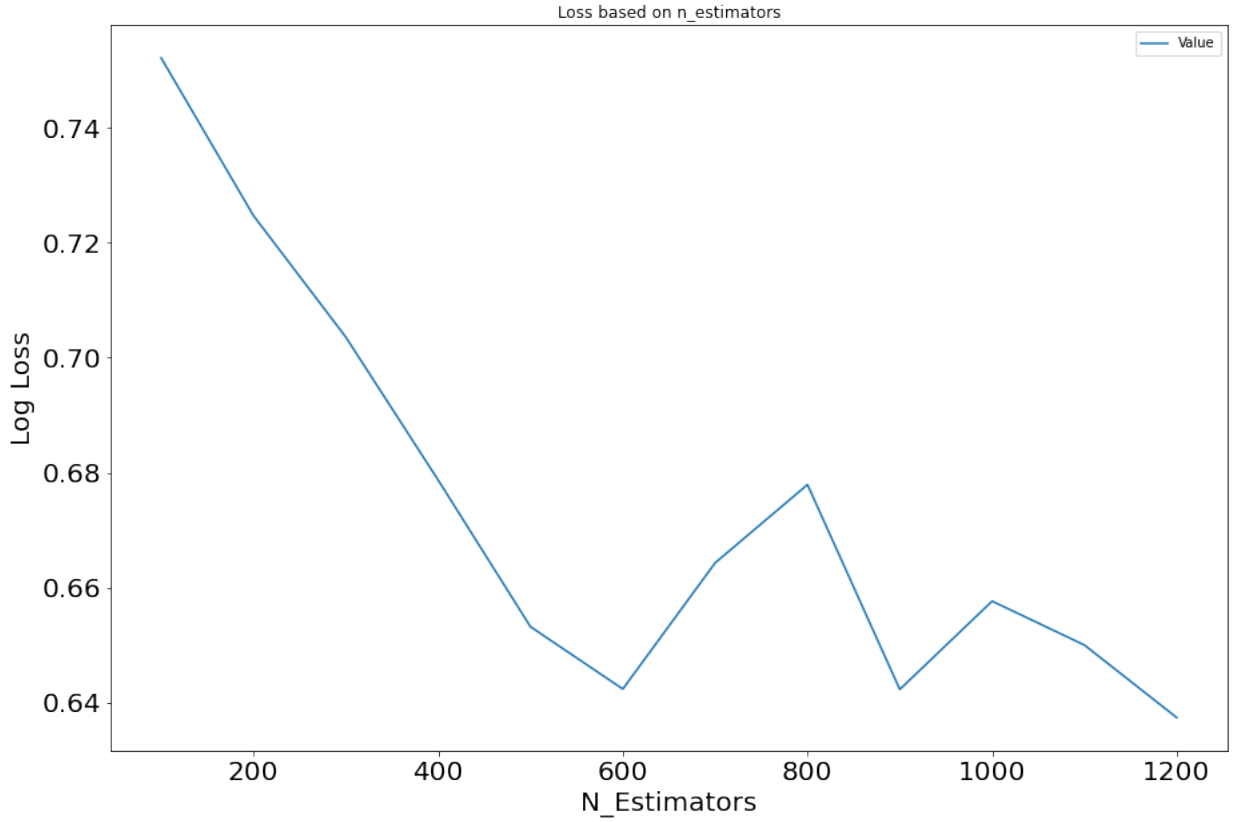

Figura 3: Variación del LogLoss a partir de $n_estimators$.

Para ejemplificar como los hiperparametros afectan la predicción final, podemos ver que este gráfico muestra la variación de Log Loss basándose en la cantidad de estimadores que utiliza el Random Forest (n estimators). A medida que aumento la cantidad de estimadores, el Log Loss va bajando hasta llegar a un limite alrededor de los 1200 estimadores donde ya el error vuelve a subir. La cantidad de arboles a utilizar se tiene que adecuar a la cantidad de datos que tenemos en nuestro set y ajustarlos también teniendo la profundidad máxima de cada árbol

3.2. Gradient Boosting

Gradient boosting, es una técnica de aprendizaje automático utilizado para el análisis de la regresión y para problemas de clasificación estadística, el cual produce un modelo predictivo en forma de un conjunto de modelos de predicción débiles, típicamente árboles de decisión.

3.2.1. XGBoost

XGBoost es una implementación especial de Gradient Boosting con la diferencia de que presenta mejoras en la velocidad de ejecución y en la performance de las predicciones.

Classification

Al igual que Random Forest inicialmente se probo este modelo utilizando XGBoost **Classification**. Pero en este caso encodeando todos los features categóricos en numéricos ya que XGBoost no puede utilizar features que no sean numéricos.

Dentro de todos los modelos de clasificación utilizados fue el que mejor resultado nos otorgo en cuanto a las predicciones. Pero aun así, los resultados obtenidos no nos eran de agrado. Dentro de nuestro set de test interno obteníamos precisiones elevadas pero a la hora de testearlo con kaggle los resultados eran errados. Entonces tras investigar como hicimos con Random Forest decidimos probar el modelo Regressor.

Regressor

En este modelo se presentó una mejora tanto en la precisión del modelo, en el Log Loss y en el RMSE. Simplemente con los hiperparámetros predeterminados ya presentaba una mejora, luego de una ardua búsqueda de dichos hiperparámetros mediante las herramientas GridSearch y RandomSearch; y mediante la utilización de los mismo obtuvimos una mejora bastante considerable tanto en el set de test interno como en kaggle. Al ver que no se obtenían mejoras buscando distintos hiperparámetros se comenzaron a utilizar distintos encodings de los datos.

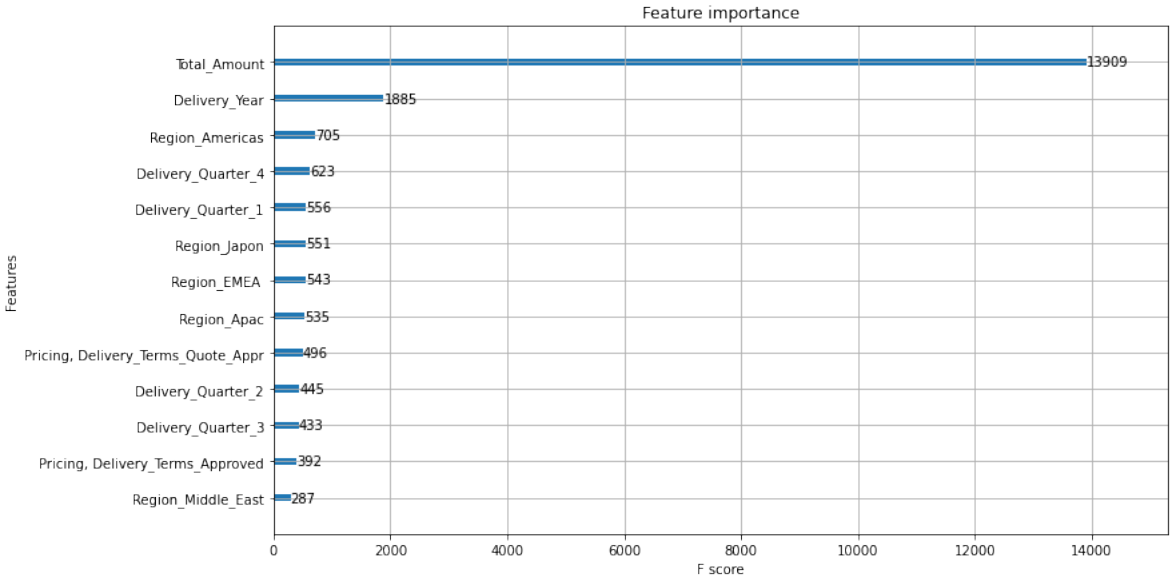

Figura 4: importancia de los features utilizando OneHotEncoder.

El primer encoding utilizado fue el OneHotEncoder. Fue uno de los encoder que utilizamos al principio de la investigación. Nos otorgaba buenos resultados, a pesar de no tener los mejores features. Se observa claramente en el gráfico que utilizábamos features que no tenían tanta importancia

A lo largo de todo el trabajo utilizamos distintos encodings y distintos features. Uno de los encodings que mejor resultado nos dio fue le Mean Encoding. Sin embargo, sabemos que utilizar este encoding en abuso puede resultar en un overfitting dado que este tipo de encoding crea una correlación directa entre el feature y el target, y esto puede cegar al modelo y darle una importancia mayor a la que ese feature deberia tener.

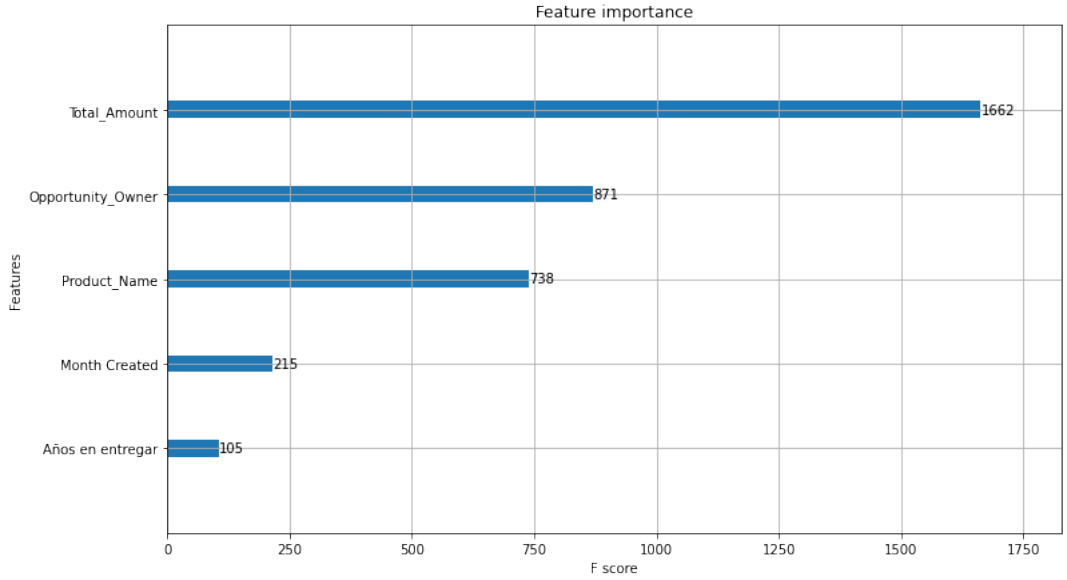

Figura 5: importancia de los features utilizando Mean Encoding.

Por ultimo observamos las predicciones y su promedio.

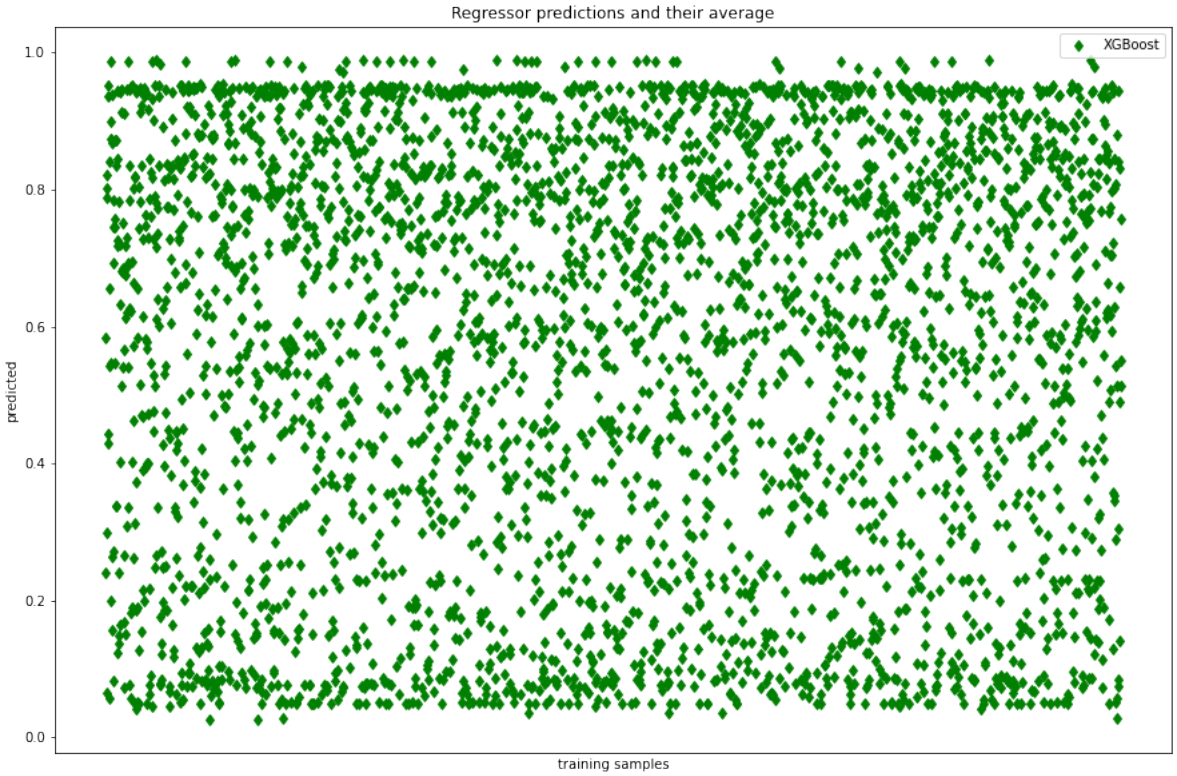

Figura 6: Predicciones y su promedio.

Observamos que hay una distribución bastante interesante a lo largo de todos los valores. Muy pocos llegan a ser una oportunidad concreta (aquellos valores iguales a 1) y también muy pocos llegan a no concretar ninguna oportunidad (aquellos valores iguales a cero).

3.2.2. CATBoost

CATBoost nos provee herramientas para hacer Gradient Boosting para problemas de clasificación y regresión. Los beneficios que ofrece CATBoost son: Ordered Boosting para preveer over-fitting y Handling nativo para features categóricos.

Esta fue una de las primeras herramientas de boosting que utilizamos y, por su cuenta, no pudo superar los resultados que obtuvimos con el Random Forest o XGBoost, algo que nos resultó extraño ya que en nuestra investigación encontramos que CATBoost suele ser mejor que los Random Forest dado el tipo de boosting que utiliza. Sin embargo, este algoritmo resultó esencial en otros métodos de ensamble que explicaremos mas adelante que nos consiguieron nuestros mejores resultados en ese momento, dada la combinación con otros modelos.

CATBoost fue el primer tipo de modelo que nos encontramos con una muy alta cantidad de hiperparámetros disponibles, por lo que estuvimos un tiempo investigando la importancia de cada uno y probando combinaciones que nos generen mejores modelos.

Utilizamos la librería Shap para analizar los features en este modelo.

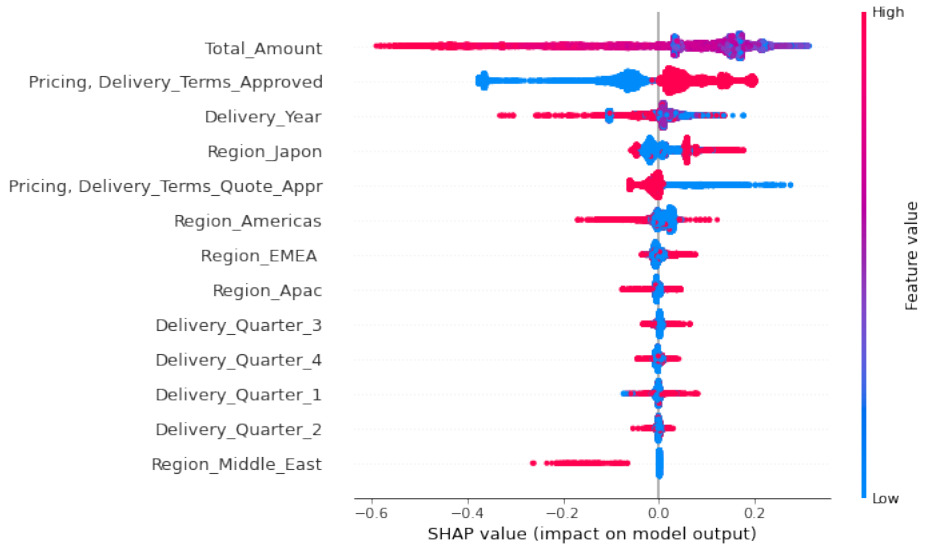

Figura 7: importancia de los features utilizando OneHotEncoder.

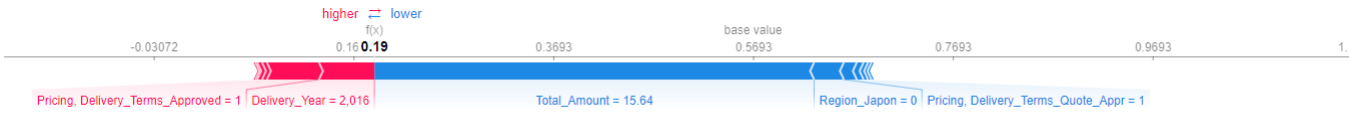

Figura 8: Primeras predicciones del modelo.

Por ultimo observamos las predicciones y su promedio.

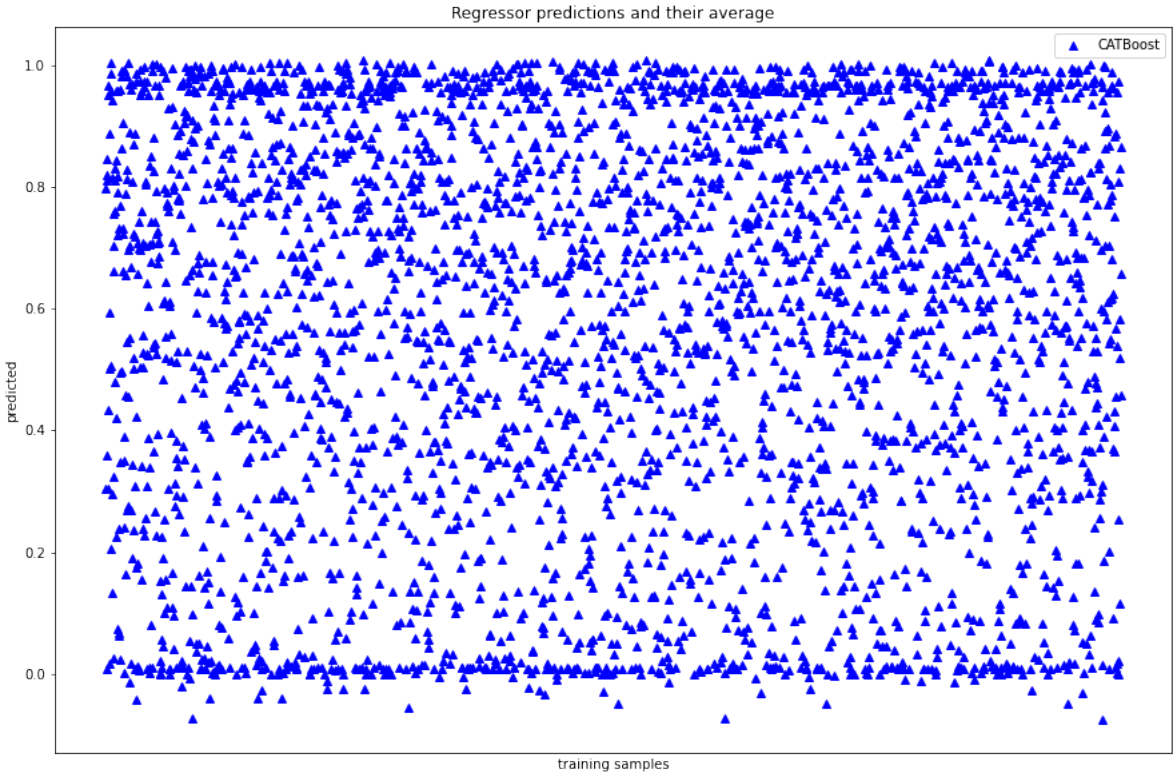

Figura 9: Predicciones y su promedio.

Al igual que XGBBoost hay una distribución bastante dispersa. Es muy parecida a la distribución del algoritmo mencionado anteriormente, observando ambas distribuciones comprendimos por qué los resultados de kaggle eran muy similares, ya que la distribución de las predicciones en su mayoría son similares.

3.2.3. LightGBM

LightGBM al igual que XGBoost y CATBoost también es una implementación de Gradient Boosting. Su principal diferencia con el anterior reside en que sus árboles tienden a crecer verticalmente. Esto genera una mejor perfomance cuando se trabaja con una gran cantidad datos pero asimismo tiende a hacer overfitting con sets de datos pequeños.

Utilizando este tipo de boosting vimos que la performance bajaba en comparación con los otros métodos de boosting, por lo que intentamos dedicarle mas tiempo ajustando sus hiperparametros pero de igual manera los resultados seguían siendo peores tanto con el set de test propio como los de Kaggle. Nuestra teoría es que el set de datos no tiene una cantidad suficiente como para que este tipo de boosting sea efectivo y es por eso que los resultados no eran lo que esperábamos.

Por ultimo observamos las predicciones y su promedio.

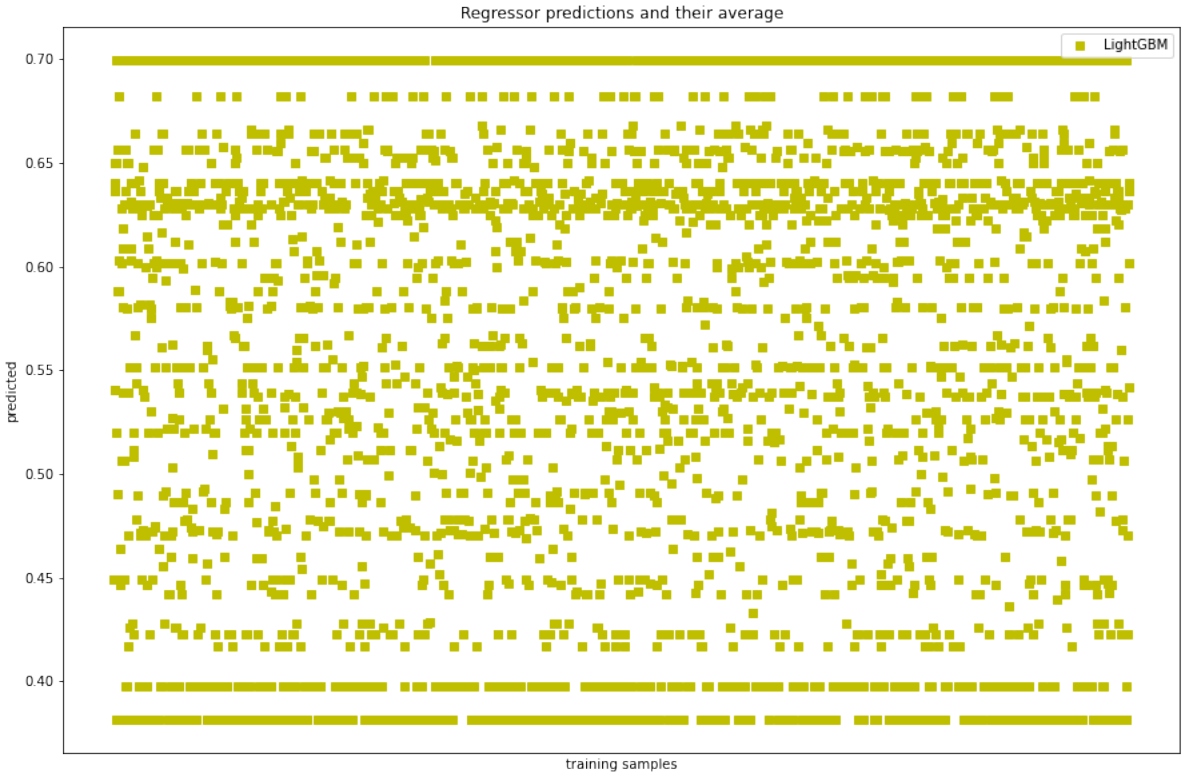

Figura 10: Predicciones y su promedio.

A diferencia de XGBoost y CATBoost esta distribución de las predicciones no hay oportunidades que tengan mucha probabilidad de concretar al venta o no. Los valores se encuentran entre 0,70 y 0,40 o un poco menos; en cambio en las demás predicciones mostradas había mucha más probabilidad de éxito o de fracaso.

3.3. KNN

KNN es un algoritmo de clasificación de Machine Learning. Este funciona de forma que predice la clase de un punto a partir de la mayoría de sus k-vecinos más cercanos. Este algoritmo no necesita tiempo de entrenamiento ya que calcula los vecinos a la hora de predecir, los tiempos de predicción son mayores a los de otros modelos.

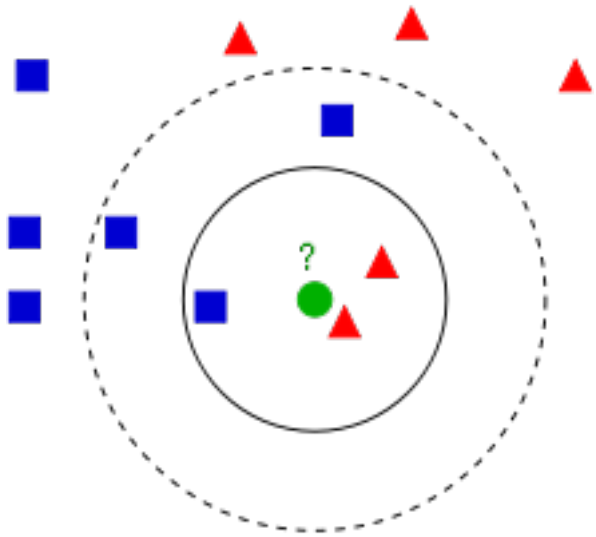

Figura 11: Ejemplo clásico de KNN.

En la figura 2 podemos observar como nuestra consulta es un circulo y los 3 (k) vecinos más cercanos son dos triángulos y un cuadrado. En consecuente, será clasificado como un triángulo.

Existe una desventaja en este algoritmo llamada el problema de la dimensionalidad, en este algoritmo a medida que aumentamos la cantidad de dimensiones se vuelve cada vez menos eficiente y necesitamos de mas cantidad de datos.

Para poder usar KNN hay que definir el valor de k, es decir, cuantos vecinos vamos a considerar. En este trabajo realizamos Grid Search para averiguar que k optimizaba el LogLoss de nuestro modelo. Comenzamos con valores de entre 2 y 6, con el KNN de Classifier, obteniendo una precisión regular. Finalmente llegamos a los resultados esperados con un k de entre 30 y 40 usando el KNN de regressor.

A continuación podemos observar uno de los tantos grid search hechos, combinando distintos hiperparámetros con igual K, para ver como es que variaban las predicciones:

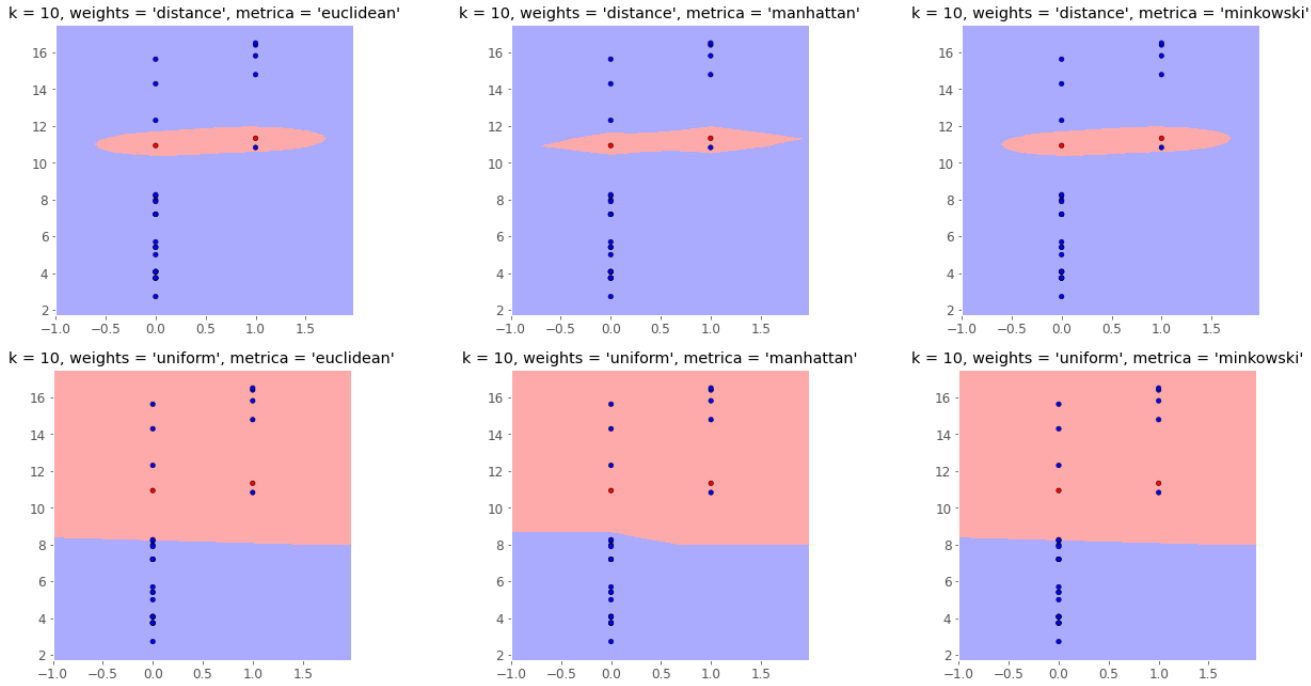

Figura 12: Variación de predicciones.

Observamos que lo que realmente cambia las predicciones es el hiperparámetro weights, sin embargo seguiremos buscando el mejor modelo.

Cuando aplicamos el KNN de Regressor vemos que el modelo es mucho mejor. Utilizando los parámetros casi por defecto, veamos como varia el error a medida que aumentamos K.

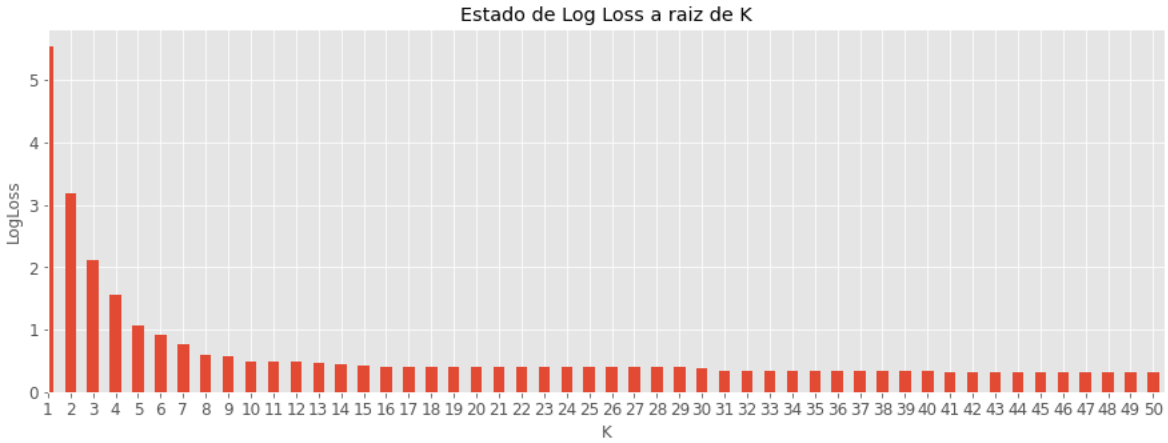

Figura 13: Variación de Log Loss.

Podemos ver que a medida que aumentamos K, el error disminuye y nuestro modelo mejora, pasando de aproximadamente 5 a un bajísimo numero de menos que 0.35. Luego de un análisis, tomamos K = 43, lo cual nos da un Log Loss de 0.3199283714757076.

Una vez entrenado el modelo sólo quedaba predecir con nuestro DF de test. Hicimos lo dicho y las predicciones quedaron de la siguiente forma:

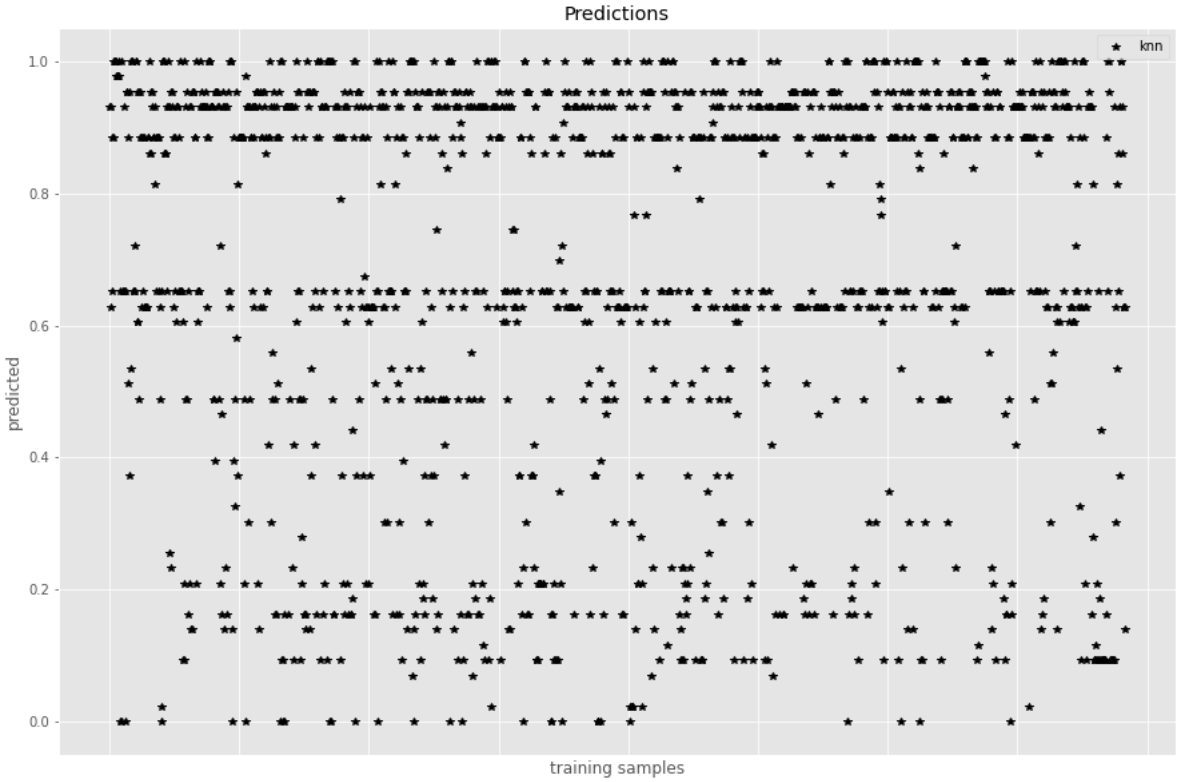

Figura 14: Predicciones finales.

Observamos que el modelo tiende a predecir valores mas cercanos a 1 que 0, esto se ve reflejado en Kaggle ya que no obtiene los resultados esperados.

3.4. Redes Neuronales

3.4.1. MLP

MLP o Multi Layer Perceptron, es un algoritmo de Redes Neuronales, que mejora al Perceptron Simple (una sola capa de neuronas). El MLP utiliza varias capas y puede resolver problemas que no son linealmente separables.

Al principio se utilizo un **MLPClassifier** de la libreria Sklearn pero esta daba un error lejos de lo esperado. Inmediatamente después se opto por usarlo con un **MLPRegressor** obteniendo de esta forma, un error mucho menor: 0,69. Probando varios de sus parámetros con **Random Search** vimos que genera *overfitting*, ya que en nuestro test interno daba un puntaje excelente y un logloss muy bajo, pero cuando le introducíamos los datos de Kaggle daba incoherencias.

Entonces optamos por agarrar una combinación de los parámetros que se utilizaron en el random search y de esta forma se obtuvo una puntuación en *Kaggle* de 0.57. Que si bien es buena supera a los otros modelos. La distribución de las predicciones son las siguientes.

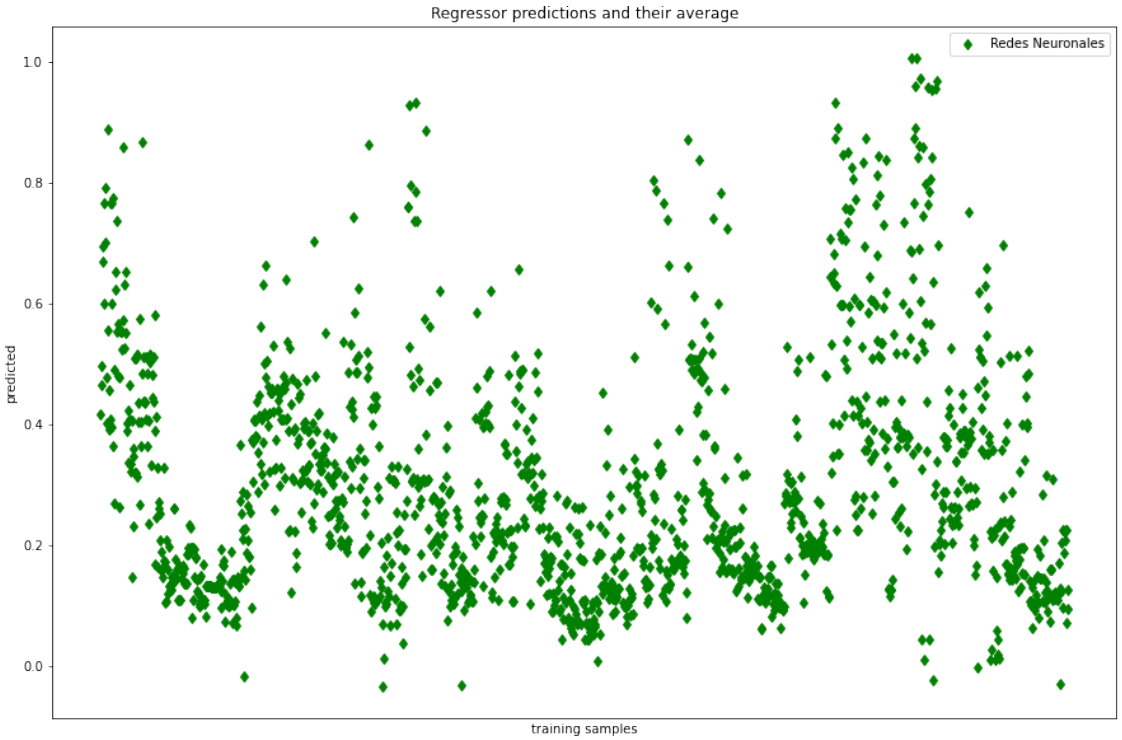

Figura 15: Distribución De Predicciones MLP

Las predicciones en promedio dan 0,29 y teniendo un std de 0,18. Por lo que vemos el algoritmo predice que las ventas fueron en su mayoría no exitosas.

3.5. Recursos Adicionales

3.5.1. Voting Regressor

VotingRegressor es una herramienta, perteneciente a la biblioteca de SKlearn, que nos permite ensamblar varios modelos de regresión, que no estén entrenados, al mismo tiempo. Luego, promedia las predicciones individuales para formar una predicción final.

Entonces, la idea es combinar conceptualmente varios modelos de Machine Learning, para que este conjunto de modelos puedan equilibrar las debilidades individuales.

Por ejemplo, un intento que se hizo fue tomar varios modelos, como el XGBoost Regressor de mejor predicción, CATBoost Regressor, Linear Regressor y RandomForest Regressor. Si bien el

score del test daba bastante bien probándolo en el Notebook, lamentablemente a la hora de subirlo a Kaggle no se reflejaba de la misma manera, para con ninguna de la combinaciones.

Por ultimo observamos las predicciones y su promedio. Para este gráfico se utilizaron los modelos de XGBoost, CATBoost y LightGBM con un set de datos en-codeado por el OneHotEncoder.

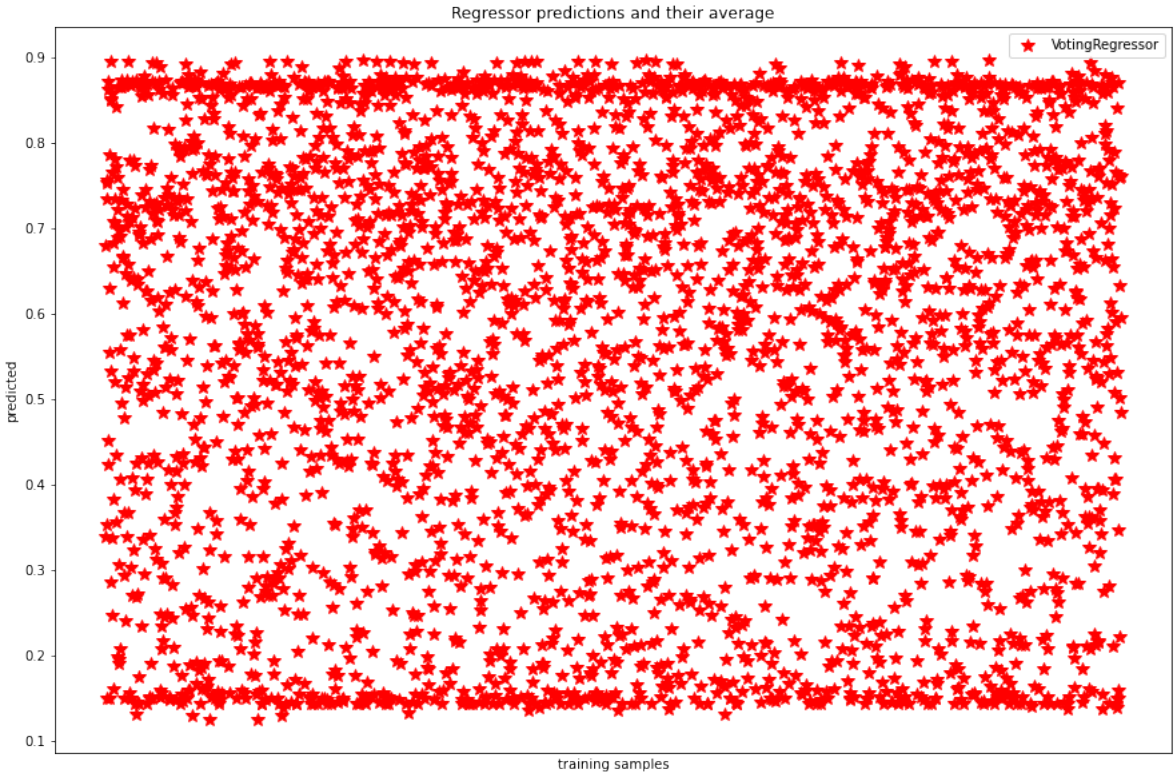

Figura 16: Predicciones y su promedio.

Notamos que es una distribución muy parecida a la de CATBoost y XGBoost. LightGBM hizo su parte, anteriormente los primeros dos modelos siempre tenían valores tendiendo a uno y/o cero. Ahora el máximo y el mínimo son 0.9 y 0.15, respectivamente. Por esta razón en nuestros Ensambles finales no utilizamos LightGBM, ya que modifica drásticamente nuestras predicciones.

3.5.2. Ada Boost

Adaptive Boosting es un meta-algoritmo de Machine Learning que reduce el sesgo y varianza. Los algoritmos Boosting se basan en el cuestionamiento: *¿Puede un conjunto de clasificadores débiles crear un clasificador robusto?*. Cabe destacar que AdaBoost es el algoritmo más popular y es quizá el más importante históricamente.

Este mismo comienza entrenando un modelo (puede ser tanto Classifier como Regressor) con el DataSet dado, y después continua entrenando copias de este modelo elegido con el mismo DataSet, pero los pesos de las instancias son ajustados acorde con el error de la predicción de cada instancia.

4. Modelo de mejor predicción

Para llegar al modelo de mejor predicción, realizamos un ensamble de los modelos XG Boost Regressor y CatBoostRegressor, a través de la herramienta Voting Regressor, y después al modelo resultado de esto le aplicamos un AdaBoost Regressor para potenciar al ensamble.

También utilizamos la combinación de varios encodings, así como One Hot Encoding, Label Encoding y Mean Encoding.

Sabemos que la utilización del Label Encoding puede ser peligrosa, ya que el modelo puede aprender de cosas inexistentes, pero lo fundamentamos con lo siguiente: De hecho es solo una columna a la que le aplicamos este encoding, la cual es 'Product Name', al principio a esta le aplicamos un Mean Encoding, pero analizando cómo iban aprendiendo los modelos, a través de gráficos, nos dimos cuenta que le daba demasiada importancia a esta columna y muy poca a las otras, lo cual resultaba raro, por eso decidimos cambiar el encoding y los modelos aprendieron de una forma más equilibrada, lo cual hizo que mejoraran los resultados de las predicciones.

Algunas columnas a las que le aplicamos el Mean Encoding son 'Region', 'Account_Type', 'Opportunity_Type'.

También se realizó la creación de una feature a través de las columnas Planned_Delivery_Start_Date y Planned_Delivery_Endt_Date, se las paso a un número entero para realizar la diferencia entre estas y se tiene una feature que represente los días planeados en los que tardaría llevarse a cabo la oportunidad. Otra cosa que implementamos fue la aplicación del logaritmo a la columna 'Total_Amount'.

A continuación veremos dos gráficos que representan la distribución de las predicciones del modelo:

La conclusión que sacamos de estos gráficos es que la distribución de las predicciones de este modelo es bastante equilibrado, aunque con una leve tendencia hacia el éxito de las oportunidades.

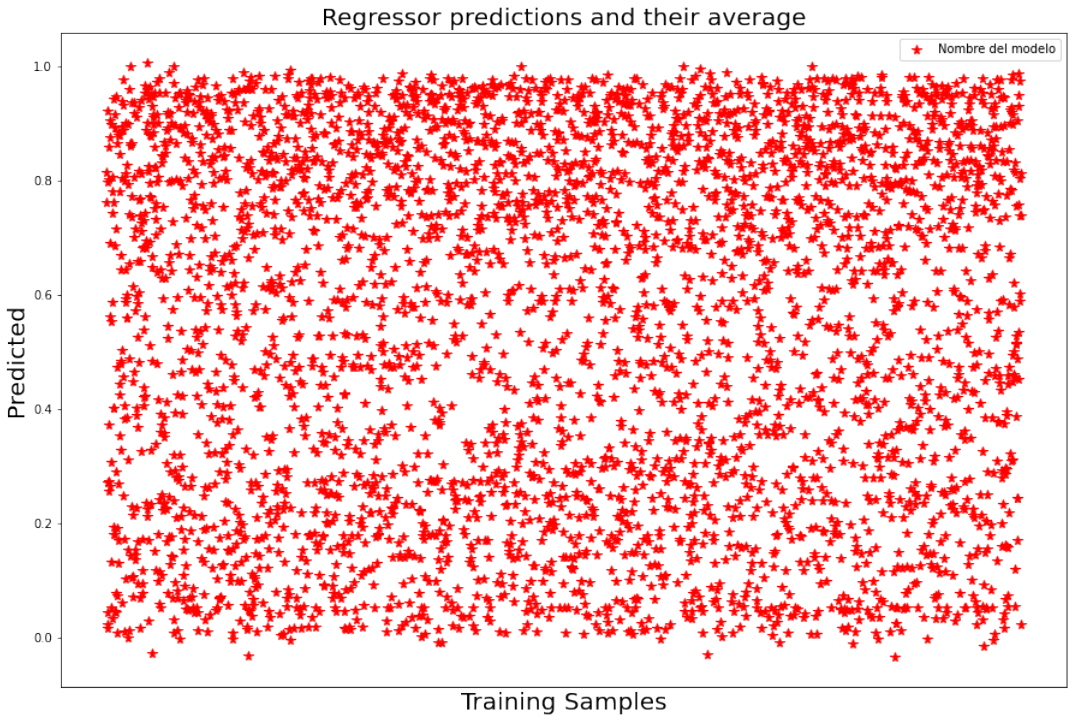

Figura 17

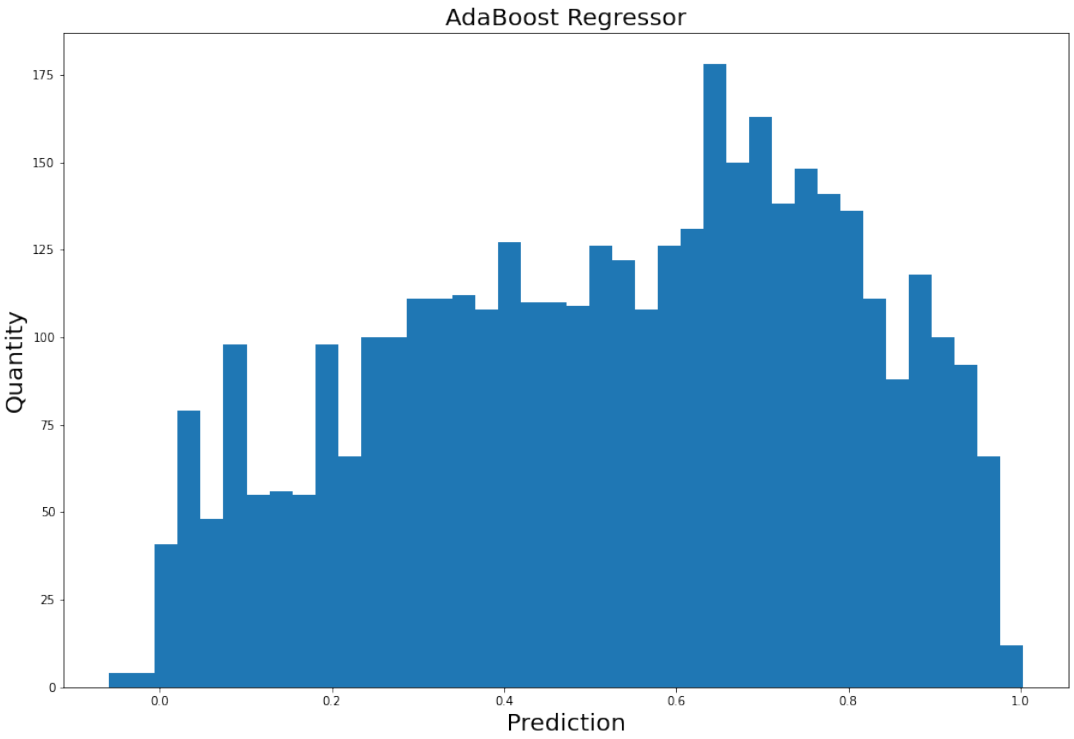

Figura 18

5. Nuestra Investigación

Inicialmente, como mencionamos anteriormente, nos decantamos por utilizar los modelos clasificadores, ya que pensabamos que dado nuestro problema lo ideal seria tener un clasificador como modelo. A pesar de una buena accuracy en nuestro set de test, nos dimos cuenta que los modelos no nos estaban dando los resultados que esperabamos en el set de Kaggle. Por lo tanto, decidimos experimentar con los modelos regresores y pensar el valor que predice como la probabilidad de exito de una oportunidad. Esto mejoro notablemente los resultados tanto en nuestro set de test propio como los de Kaggle, por lo que sentimos que estos modelos se adaptan mejor al problema, especialmente dado la forma en la que se puntua el modelo.

A lo largo del trabajo practico, investigamos sobre los distintos hiperparametros que ofrecia cada modelo y como estos afectaban las predicciones que hacian. Los metodos de seleccion de modelos que nos provee sklearn, como el Random Search y el Grid Search nos ayudaron fuertemente a la hora de elegir que modelo era el mejor para nuestros datos ya que podiamos tener un set de posibles hiperparametros y el Grid Search intentaba todas las combinaciones posibles, los que nos ahorro mucho trabajo manual.

Ademas, investigamos sobre como los distintos boostings (xgboost, catboost, etc) afectaban el resultado final de nuestras predicciones. Nos parecio particularmente interesante el caso del Ada Boost ya que nos deja elegir facilmente el modelo base del cual parte y nos permitio probar varias combinaciones de modelos que nos resultaron bastante buenos.

Por ultimo, investigamos sobre los distintos tipos de ensambles y como tomaban sus decisiones a partir de los modelos que teniamos. Tras probar con varios, nos decantamos por utilizar el Voting Regressor como el ensamble final ya que era el que nos daba las mejores predicciones sobre el set de test de Kaggle.

5.1. Librerías

Durante el Trabajo Practico utilizamos las siguientes librerias: ChefBoost, SKlearn, XGBoost, CatBoost, LightGBM, Pandas, Seaborn, Numpy, Shap y Matplotlib.

5.2. Hiperparámetros

La búsqueda de hiperparametros depende de los datos o del algoritmo a utilizar. En nuestro caso utilizamos un **Grid Search** para probar distintas combinaciones de hiperparametros y luego entrenar a nuestro algoritmo con el mejor resultado. Además, como mencionamos en la investigación, tuvimos que averiguar como afectaba cada hiperparámetro al modelo para elegir los mejores candidatos y probarlos con el Grid Search. Sabemos que esta opción es costosa en tiempo pero asegura la combinatoria de todos los hiperparametros que queremos probar.

Otra forma de búsqueda utilizada fue el **Random Search** que gana en cuestión de tiempo y prueba la combinación random (de ahí su nombre) de todos los hiperparametros dados. Pero dicha forma de búsqueda no fue utilizada en nuestros últimos modelos ya que en ciertas ocasiones no presentaban mejoras tanto en nuestro test local como en kaggle; solo fue utilizado para los primeros modelos trabajados y como alternativa del GridSearch.

6. Conclusión

A lo largo de este trabajo, se han puesto a prueba múltiples metodologías con el fin de elaborar un robusto modelo de Machine Learning capaz de predecir correctamente la naturaleza de cada oportunidad. Este trabajo nos sirvió además para tener experiencia de primera mano en la utilización de esta ciencia en un uso cotidiano y en su uso en la ciencia de datos. No solo esto sino que también, este tp sirvió para darnos cuenta lo complejo que es el mundo del Machine Learning, no basta solo con usar un algoritmo sino todo lo que hay por detrás, las distintas técnicas de Feature Engineering o de busca de hiperparámetros.

Además, notamos que ante mínimos cambios, ya sea de los valores de los hiperparametros o el agregado de un feature, el score empeoraba o mejoraba de forma notable, demostrando el efecto avalancha. Cambios que creíamos importante terminaban empeorando su comportamiento o cambios muy chicos benefician más de lo esperado.

Por otro lado, consideramos que aún quedarían distintos algoritmos y búsqueda hiperparametros que nos hubiese gustado realizar que no hemos podido, ya sea por falta de tiempo o porque lo hayamos intentado erróneamente.

7. Github y Video

Dejamos el enlace al Github del TP de Machine Learning. https://github.com/BautistaXifro/Machine-Learning-Datos

Link al video grupal. https://youtu.be/xhlcsteBJBE