UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

[75.06/95.58] ORGANIZACIÓN DE DATOS

Curso 1: Argerich

1ER CUATRIMESTRE DE 2021

TRABAJO PRÁCTICO N°2: Machine Learning

GRUPO: Pandas en júpiter	
APELLIDO, Nombres	N° PADRÓN
MASTRICCHIO, Facundo Rodrigo	100.874

Link al repositorio: https://github.com/FacuMastri/tps-orga-de-datos

Link al video: https://drive.google.com/file/d/1q713CW4RN7IjTbeqF8kQdbES1ku-FDu6/view

Índice

1. Introducción

El objetivo principal del trabajo es predecir la variable *damage_grade* que representa el nivel de daño recibido por una edificación en el contexto del terremoto que azotó a Nepal en el año 2015.

La realización del trabajo se produce con algoritmos de Machine Learning, una disciplina que busca poder generar clasificaciones en base a un entrenamiento sobre información pasada o actual, seguida de una validación de las predicciones generadas. En el trabajo se prueban distintos algoritmos, los cuales todos en distinta manera hacen uso de los datos (en particular, de sus atributos). Por lo que es muy importante saber que datos usar, y buscar como codificarlos de tal forma que mejor se aprovechen por cada uno de los modelos.

Con dicho propósito se utilizan como base tres sets de datos brindados por DrivenData. En un primer lugar el archivo *train_values.csv* que contiene diversa información sobre las edificaciones desde por cuales tipo de materiales están constituidas hasta cuantas familias habitan en ellas; el segundo archivo *train_labels.csv* que contiene el nivel de daño recibido por las edificaciones del archivo *train_values.csv*; y por último el archivo *test_values.csv* que tiene una estructura similar a *train_values.csv* y es sobre éste conjunto de datos sobre el cual debemos predecir el *damage_grade.*

2. Organización del trabajo

Para un desarrollo un poco más cómodo, se modularizo el trabajo en distintos notebooks de Jupyter. Si bien en algunas ocasiones se repitió código, el haber estructurado el trabajo de la siguiente manera nos permitió trabajar de una manera más fácil y sin tanto acoplamiento.

Los notebooks en orden de lectura y de corrida son:

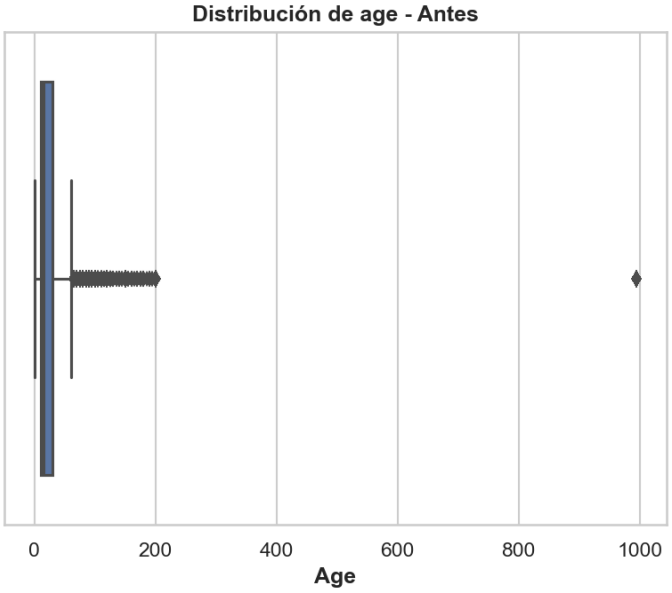
1. **Limpieza de datos:** este notebook concatena los datos del set de train y de test para realizar un análisis cuantitativo referido a los outliers principalmente de aquellas features numéricas. También se busca detectar si hubo algún error de carga de los datos al ingresar una edificación más de una vez. Genera los archivos *train_cleaned.csv* y *test_cleaned.csv* que se utilizan como base en la etapa de feature engineering.
2. **Feature engineering:** en este notebook se agregan todos los features que se consideran que pueden ser pertinentes para los modelos, así como también el tipo de encoding que nos permita sacar el mayor provecho de las variables categóricas. No se hace distinción a qué modelo se va a alimentar, si no que pasamos la responsabilidad de la selección de features al propio modelo. Este notebook genera los archivos *to_train_01.csv* y *to_test_01.csv*.
3. **Modelo particular:** cada modelo tiene su notebook particular, por ejemplo, *LightGBM.ipynb*. En cada notebook de cada modelo, se seleccionan las features que se crean más útiles o convenientes basándonos en el feature importance del propio modelo; se entrena los modelos con la selección de features efectuada y posteriormente se genera el submit para la competencia.
4. **Parameter tuning:** tres de los modelos utilizados cuentan con un notebook cuya función es la búsqueda automatizada de hiperparámetros óptimos, por ejemplo, *LightGBM-tuning.ipynb*. En particular, se optó por el uso de *GridSearchCV* basándonos en la documentación técnica de cada modelo y de diversos artículos que se pueden hallar en internet. La dinámica consistió en hallar los mejores hiperparametros hasta el momento, anotarlos y actualizar el notebook correspondiente del paso tres.

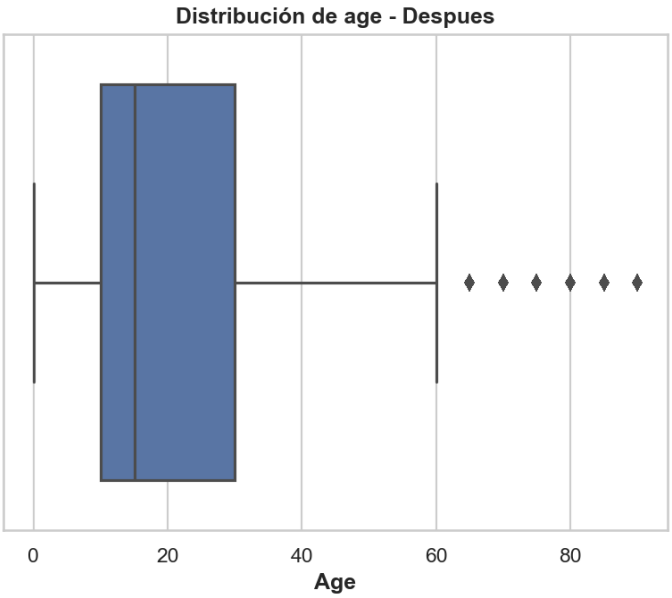
3. Features

3.1 Limpieza de datos

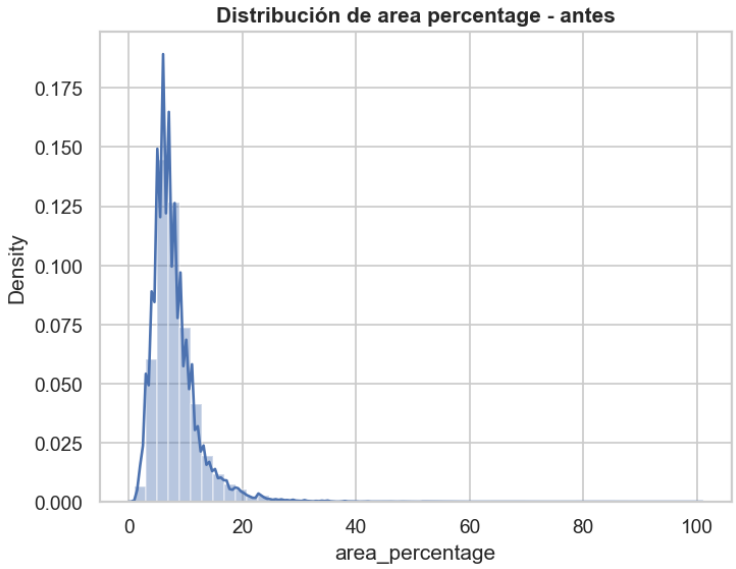
El primer paso del trabajo consistió en realizar una limpieza de datos tanto al set de train como al set de test. La limpieza tenía como objetivo encontrar posibles entradas duplicadas, ver la distribución de aquellas features numéricas y la detección de posibles outliers de dichas features que pueden llegar a molestar o no a un determinado modelo.

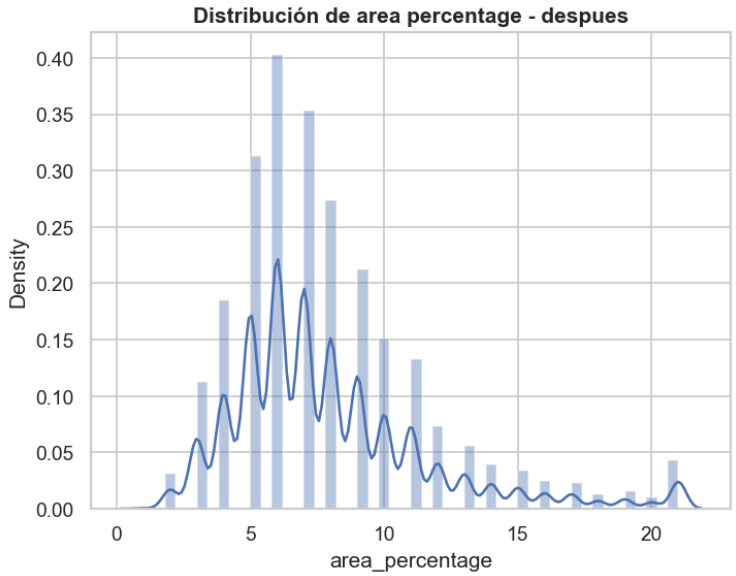
¿Cuál fue el mecanismo para detectar outliers? [1] Usamos el método de boxplot con su intercuantil. Si un determinado valor es mayor a 3 * IQR (*interquartile range*) por arriba del tercer cuantil, entonces el valor es considerado un outlier. De la misma forma, si un valor es menor a 3 * IQR por debajo del primer cuantil, entonces el valor va a ser considerado un outlier. A continuación, se muestran dos ejemplos de aplicación de esto sobre las features *'age'* y *'area_percentage'*.





[1] https://www.listendata.com/2015/01/detecting-and-solving-problem-of-outlier.html





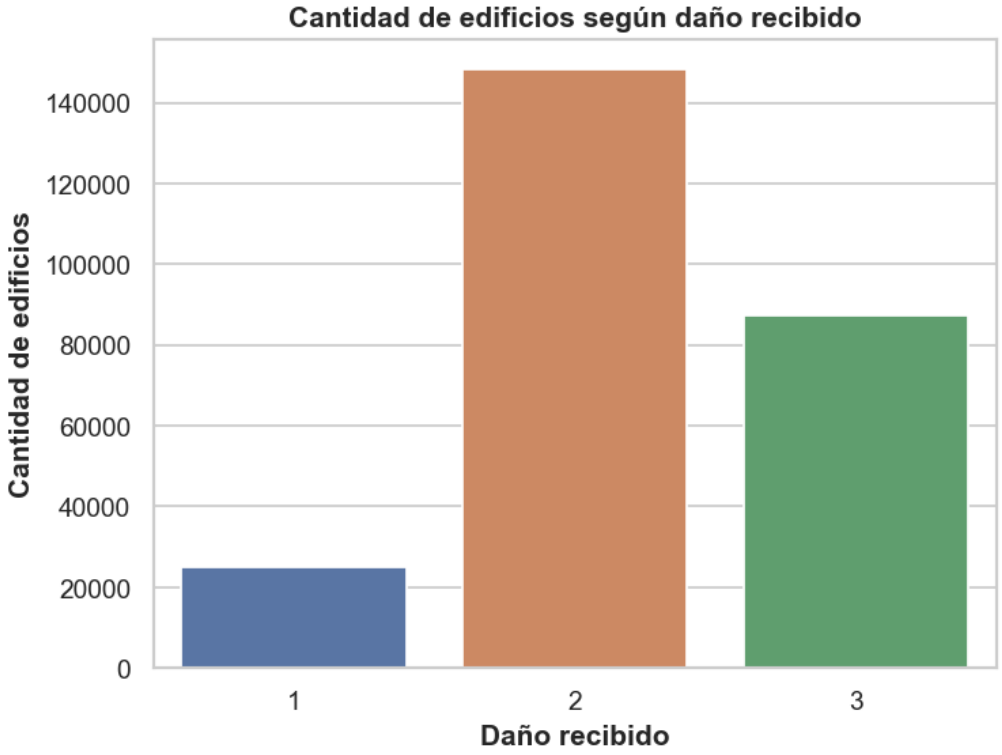
Notar que en el caso de *área_percentage*, en un principio, se da el fenómeno de *right-skewed distribution* y que luego de aplicar la transformación mencionada obtenemos una distribución algo similar a una distribución normal al reducir el rango de valores posibles que puede tomar la variable. Si bien este problema no afecta a los modelos basados en árboles, si es un potencial inconveniente para otros tipos de modelos.

¿Para las features categóricas también se aplicó el mismo método? No, para estas features solo se listó la cantidad de valores para cada categoría de las features para tener una noción de qué se podría hacer en materia de encoding. La idea de esto es que en base a esta información en el notebook de feature engineering se probaran distintos tipos de encoding que ataquen este problema de la mejor manera.

Luego de la serie de pasos y búsquedas mencionadas, se recopilo la siguiente información:

- No hay errores de cargas en el dataset. Existen varias edificaciones con exactamente el mismo valor para cada una de las features iniciales, pero tienen un distinto *building_id* como uno sospecharía.
- Existen tipos dentro de cada feature categórica cuya cantidad de muestras es muy superior comparados con la suma del resto de los tipos para esa misma categoría.

- La cantidad de edificios según daño recibido se encuentra muy desbalanceada. En un momento estuvimos tentados en hacer un *over/under sampling* para que los modelos tengan la misma cantidad de data por cada nivel de daño, y si bien algunos modelos tienen hiperparámetros para combatir esto, ninguna era una buena solución. El problema principal de estas soluciones era que el set de test (el que predecimos para hacer el submit) también proviene de la misma distribución desbalanceada que el set de train, por lo que probablemente al buscar aumentar la cantidad de predicciones correctas para el tipo de daño 1 nos generaría disminuir la cantidad de predicciones correctas para el tipo de daño 2.



3.2 Feature engineering

En esta sección, se busca todo tipo de atributo que pueda resultar útil a los modelos a utilizar. Cabe destacar que, si bien se mencionarán varias nuevas features, no todas fueron utilizadas de forma constante. Es decir, una vez que descubrimos que no aportaban nada (o incluso peor, generaban *overfit*) las descartábamos.

3.2.1 Trabajando con los geo_level_ids

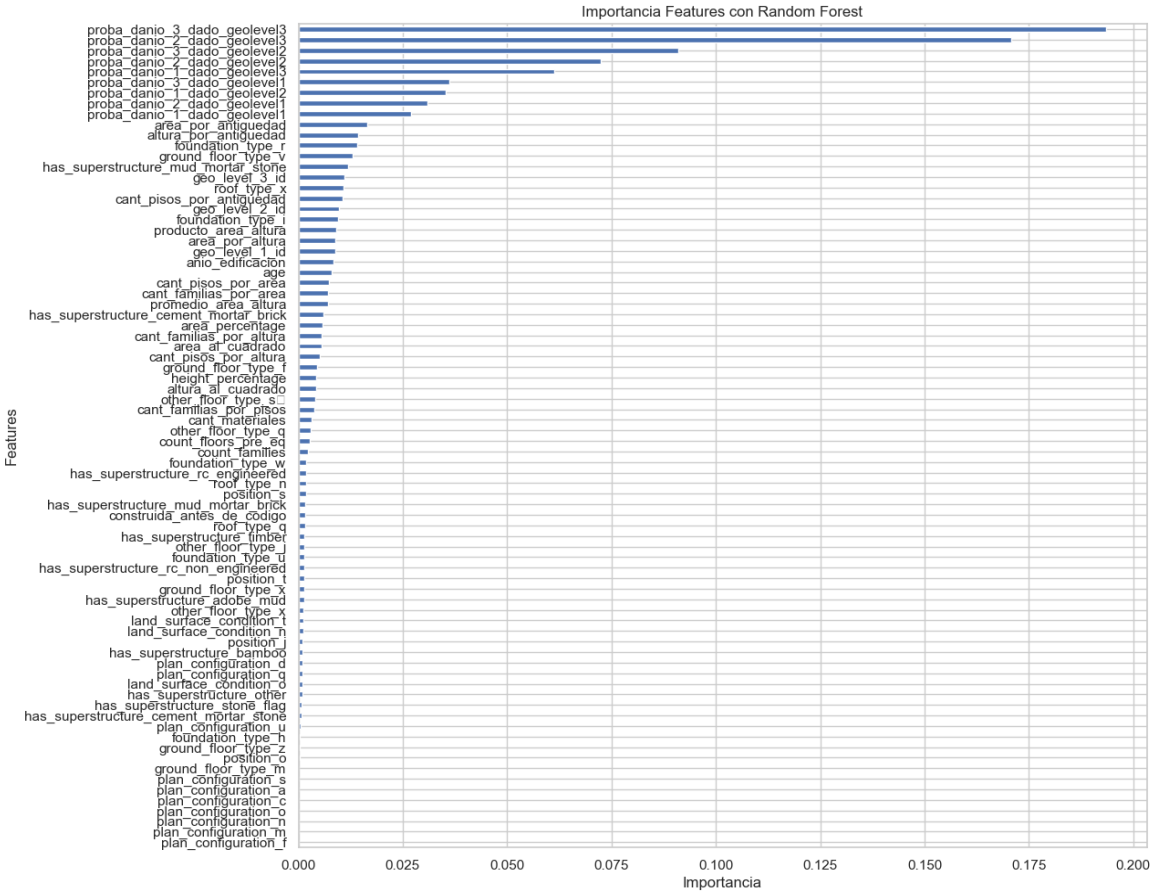
3.2.1.1 Enfoque de probabilidad condicional

Del TP1 sabemos que los *geo_level_ids* tienen una alta cardinalidad para los tres tipos de IDs, pero además si usáramos los IDs tal y como están podría pasar que introdujéramos un cierto 'orden' que no represente lo que en realidad significan esos valores generando una posible molestia a los modelos. Por esto, buscamos aplicar probabilidad condicional de manera de obtener, por ejemplo, P(damage_grade = 1 | geo_level_1 = 14) . Por ende, por cada columna de geo_level_id obtendríamos 3 nuevas features. El potencial problema de esto es que estaríamos filtrando información tanto al set de validación a usar para medir el score de nuestros distintos modelos, como al set de test al cual debemos de predecir las labels para DrivenData. Inclusive, en algunos casos hay IDs que tienen muy pocas o nulas muestras generando que esta alternativa sea una fuente importante de *overfitting*. Por ejemplo, para el *geo_level_1_id* se puede ver que

cada columna normalizada suma 1, y que además el tipo 2 de daño es el tipo que tiene mayor probabilidad.

geo_level_1_id	0	1	2	3	4	5	6	7	8	9	...	21	22	23	24	25	26	27	28	29	30
damage_grade																					
1	0.08	0.15	0.09	0.03	0.04	0.17	0.09	0.05	0.03	0.14	...	0.02	0.13	0.06	0.21	0.08	0.35	0.04	0.00	0.02	0.09
2	0.77	0.73	0.66	0.60	0.77	0.75	0.67	0.59	0.45	0.69	...	0.39	0.74	0.69	0.69	0.78	0.56	0.48	0.59	0.88	0.79
3	0.15	0.11	0.25	0.36	0.20	0.09	0.25	0.35	0.52	0.17	...	0.58	0.13	0.25	0.10	0.14	0.09	0.48	0.41	0.10	0.11

Al momento de utilizar esta alternativa, no éramos conscientes del nivel de *overfitteo* que generaba esta forma de encarar los geo level IDs. Al utilizar nuestro primer modelo, Random Forest, nos trajo muy buenos resultados ya que mejoraba bastante el score en aproximadamente 0.0050. Sin embargo, el score para tanto el set de train como de validación local nos mostraban una clara tendencia al overfitting como se muestra en el siguiente grafico de importancia de features para Random Forest en una de nuestras primeras iteraciones.



Por esto, y con la esperanza de encontrar una alternativa que no implique utilizar el target label, nos encontramos con la siguiente opción.

3.2.1.2 Enfoque de feature embedding usando redes neuronales

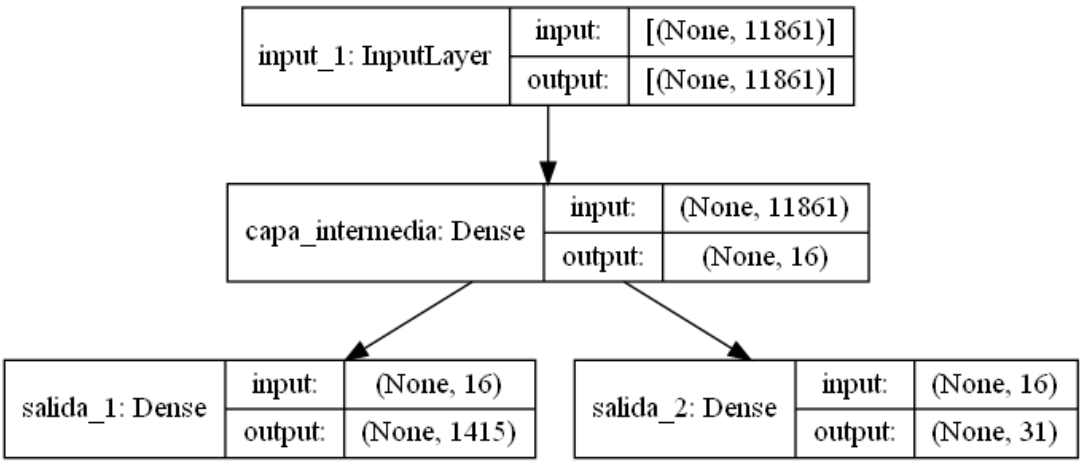
Leyendo artículos[2] [3] sobre competencias de ML nos encontramos con una técnica que aparentemente es muy usada, en especial con datasets donde la cardinalidad de las features categóricas es muy grande. Consiste en usar *embedding* similar al que se usa para textos

[2] Why you should always use feature embeddings with structured datasets

[3] Understanding entity embeddings and its application

(Word2Vec) pero con redes neuronales para casos categóricos numéricos. Un embedding representa una data categórica en un vector de una dimensional menor a la feature que se está analizando. Por ejemplo, si la feature es de R^N entonces el mappeo de embedding lo hace de $R^N \rightarrow R^M$ el cual es un espacio de menos dimensiones. Es decir, actua como si fuera un encoder.

¿Cómo se realizó este embedding? Lo realizamos utilizando la API de *Tensorflow (Keras)* creando una red neuronal con una capa de input, una única capa intermedia, y dos capas de salidas cuyas funciones de activación fueron la función *sigmoide*. En la capa de input se recibían los IDs del nivel 3 de *geo level*, mientras que en las capas de salidas se esperaban los IDs de los niveles 1 y 2. La capa intermedia nos definía cuantas features (columnas) podíamos llegar a generar a partir de la cantidad de neuronas que ésta iba a tener. Entendemos que la cantidad de neuronas es un hiperparámetro y es por esto por lo que en un principio utilizamos solamente diez neuronas, número que en iteraciones posteriores fue reemplazado por dieciséis ya que vimos resultados positivos. Pasado esta cantidad de neuronas (y por ende de features) no se veían aumento sustancial en los scores.



¿Pero y esto que tiene que ver con un encoding? La idea se basaba en extraer el output de la capa intermedia. Keras trae una función especialmente que se encarga de esto, así que solamente tuvimos que encargarnos del mappeo de este output hacia los set de train y de test. Existen otras alternativas en donde se incluyen otras features en la red neuronal de manera de combinar y no diluir la mayor cantidad de features posibles, sin embargo, esto es algo que nos queda pendiente. Como todo este proceso fue costoso, se persistió los pesos de la red neuronal y se generaron nuevos archivos *train_geo_embedded.csv* y *test_geo_embedded.csv* de manera de concatenar los nuevos datos adquiridos en futuras iteraciones.

Desde el punto de vista del score para la competencia, nos genera una breve mejora con respecto a la alternativa anterior (~0.003). Pero el mayor impacto viene de la mano de la reducción del overfit, debido a que ajustando un poco los hiperparámetros de los modelos y usando esta alternativa, se logró tener un testing score local y un DrivenData score muy similar.

Sin embargo, desconocemos si tiene una utilidad real productiva. Por ejemplo, ¿que representan los geo level embedded para un modelo? ¿Tiene sentido conocer la ubicación de una casa a la hora de saber cuánto daño va a recibir por un terremoto? ¿Tenemos en cuenta si la casa se encuentra en el epicentro del terremoto o si está más alejada?

3.2.2 Features basadas en geo level id

[cant_geolevel1, cant_geolevel2, cant_geolevel3, relacion_geolevel1_geolevel2, relacion_geolevel1_geolevel3, relacion_geolevel2_geolevel1, relacion_geolevel2_geolevel3, relacion_geolevel3_geolevel1, relacion_geolevel3_geolevel2]

Los atributos refieren al conteo de *geo level IDs* vistos por cada valor dentro de los niveles 1, 2 y 3. Además, se agregó la relación (división) entre estas sumas halladas para los distintos niveles que se presentan en el set de datos. En esta sección se generaron 9 nuevas features.

3.2.3 Features basadas en la antigüedad del edificio

[anio_edificacion, construida_antes_de_codigo]

Se codifica el año de edificación de los edificios. Como sabemos que el terremoto fue en 2015 y que los datos recolectados fueron en ese mismo año, podemos calcular en qué año fue construida cada edificación. Al mismo tiempo, del TP1 sabemos que en 1994 se creó el código de construcción que establecía de qué manera debían construirse las viviendas para que sean resistentes a terremotos.

3.2.4 Features basadas en cantidad de pisos, área, altura y cantidad de familias

En este apartado se generaron diversas features y de distinta índole. Por un lado, se crearon 28 nuevas features relacionados con la diferencia o desviación por cada variable categórica y por cada atributo numérico respecto del promedio de la categoría. Por ejemplo, para la variable foundation_type y los atributos age, area_percentage, height_percentage, count_floors_pre_eq tenemos: *[diff_prom_foundation_type_age, diff_prom_foundation_type_area_percentage, diff_prom_foundation_type_height_percentage,*
diff_prom_foundation_type_count_floors_pre_eq]

De la misma manera, se generaron 28 nuevas features, pero referidos a la desviación respecto del máximo y 28 nuevas features referidas a la desviación respecto del mínimo para cada categoría por cada atributo numérico.

En cuanto a interacción uno a uno entre features, se crearon features basadas en la cantidad de pisos hallando la relación entre ésta feature y la antigüedad, el área, y la altura. Siguiendo la misma idea se utilizó el área y la altura relacionándolas con la antigüedad. Al mismo tiempo, se calculó el promedio entre el área y la altura; y se halló el producto entre el área y la altura. En resumen, las features nuevas de este párrafo fueron 13 y fueron las siguientes: *[cant_pisos_por_antiguedad, cant_pisos_por_area, cant_pisos_por_altura area_por_antiguedad, altura_por_antiguedad, area_por_altura, producto_area_altura, area_al_cuadrado, altura_al_cuadrado, promedio_area_altura, cant_familias_por_pisos, cant_familias_por_area, cant_familias_por_altura].*

Finalmente, de esta sección se desprenden un total de 97 nuevas features para nuestro set de datos.

3.2.5 Features basadas en tipos de estructuras

En esta parte solamente se creó una única nueva feature llamada *['cant_materiales']* que hace referencia a la cantidad de estructuras que posee una vivienda.

3.2.6 Agregando clusters encontrados por K-Means

En una de las iteraciones que realizamos, se decidió por probar K-Means junto con el mejor algoritmo que nos había dado resultados hasta el momento el cual era LightGBM. La idea era bastante directa: calcular a cuál clúster pertenece cada registro y agregarlo como una feature que luego debíamos encodear. Lamentablemente, no nos dio buenos resultados y creemos que se debe por la cantidad de clústers que nosotros tomamos (seis). Es algo que deberíamos haber seguido probando con valores un poco más alto y comparar resultados.

3.2.7 Concatenando los tipos de estructuras

Una pregunta que nos hicimos fue ¿Cómo podemos aprovechar todas las columnas booleanas que nos indican que tipos de estructuras tiene una edificación? Viendo que cada columna booleana tenía poco peso entre sí, la respuesta que se nos ocurrió fue la de comprimir toda esa información en una única feature. Esta columna va a tener como posibles valores un string con el nombre de todos los tipos de estructuras que tiene una edificación. Posteriormente, se encodea esta nueva feature con target encoding para que pueda ser utilizada por los modelos. Desafortunadamente no nos dio buenos resultados, de hecho, tuvimos un retroceso en el score obtenido hasta ese momento. Creemos que esto se debe a que se pierde mucha información si se decide juntar todas las features en una sola columna.

Nos hubiera gustado probar algunos de los algoritmos de reducción de dimensiones que se ven en la materia, sobre todo porque hay una gran cantidad de columnas booleanas que, a priori, no parecen ser importantes pero que son una buena posibilidad para el minado de información.

3.2.8 Encoding de variables categóricas

3.2.8.1 One-Hot encoding

En las primeras iteraciones utilizamos el conocido (y mal utilizado) One-Hot Encoding ya que nos proporcionaba una manera rápida de probar nuestros modelos. Como es conocido, genera una situación de overfit importante teniendo en cuenta que, en nuestro caso, nos creaba 34 nuevas columnas booleanas considerando que:

- La columna foundation_type tiene 5 tipos de valores posibles.
- La columna ground_floor_type tiene 5 tipos de valores posibles.
- La columna plan_configuration tiene 10 tipos de valores posibles.
- La columna position tiene 4 tipos de valores posibles.
- La columna legal_ownership_status tiene 4 tipos de valores posibles.
- La columna other_floor_type tiene 4 tipos de valores posibles.
- La columna land_surface_condition tiene 3 tipos de valores posibles.
- La columna roof_type tiene 3 tipos de valores posibles.

Por lo tanto, y considerando que el training score y test score locales se diferenciaban mucho entre sí, y el test score local no era cercano al score obtenido en DrivenData, se procedió a buscar una nueva opción que no nos generara tantas columnas (idealmente que no nos generara ninguna nueva).

3.2.8.2 Target encoding

Como vimos en las clases teóricas, este tipo de encoding es bastante fácil de usar porque genera valores que pueden explicar el target. No obstante, tiene un gran downside el cual es el overfitting que puede a llegar a generar. Si no se le aplica cierto smoothing caemos en el mismo problema que describimos en la sección de los geo level IDs a la hora de aplicar probabilidad condicional. En la versión clásica de este encoding, existe un hiperparametro que regula el smoothing y es el cual utilizamos para reducir el overfit. En principio, tomamos un valor de 300 que se puede interpretar como que se necesitan alrededor de 300 muestras de una determinada categoría para que el promedio de dicha categoría prevalezca sobre el promedio global. Dicho valor lo tomamos en base de analizar cuantas observaciones tenían cada tipo de cada variable categórica. Algo que nos hubiera gustado intentar es variar este hiperparámetro y ver cómo afecta a los modelos y al score en general.

Este encoding fue el que mejor nos resultó y tuvo un impacto inmediato. Nos proporcionó una mejora de ~0.03 en el score. Intentamos mezclar el target encoding y el one-hot encoding

al aplicarlo a distintas features categóricas, pero no resultó mejor que aplicar target encoding a todas ellas.

¿Aplicaron target encoding a las columnas de geo level id? Si, fue una de las pruebas que realizamos, pero no nos trajo mejor resultados comparado con la alternativa descripta usando una red neuronal.

3.2.9 No subestimar una función de split

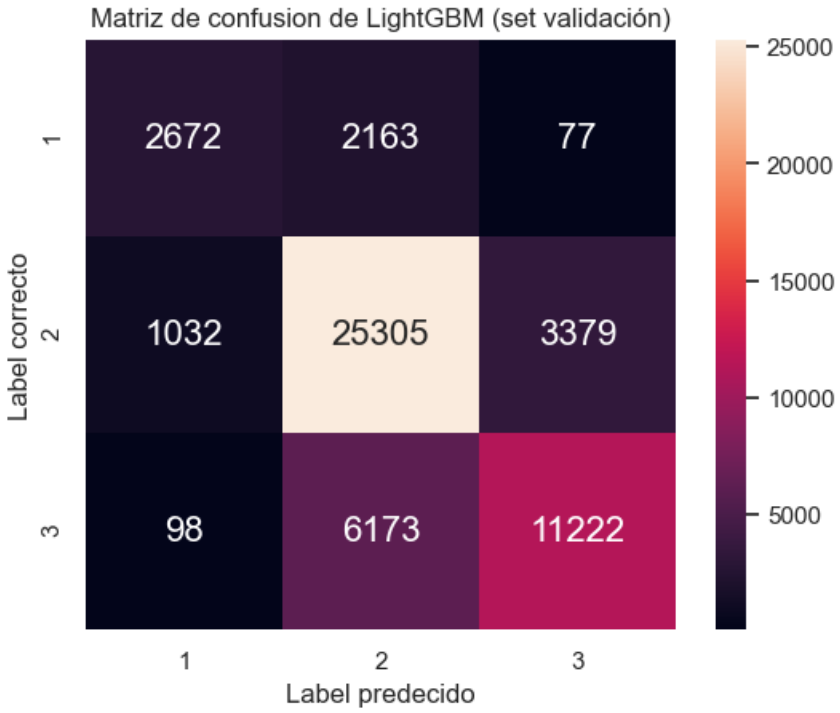
En las primeras pruebas que fuimos realizando dos cosas en particulares nos llamaron la atención:

1) El claro overfitting que estábamos realizando por no tunear los hiperparámetros de los modelos y por agregar features muy específicas con información set de train.
2) Al realizar múltiples corridas de un mismo modelo, con las mismas features y los mismos hiperparámetros, no obteníamos un resultado constante.

El ítem uno es el que tenía una solución muy directa y por eso no nos preocupaba tanto ya que tarde o temprano tendríamos que realizar una selección de features y tunear los hiperparametros de lo modelos. Pero el ítem 2 nos generó algo de confusión ¿Cómo es que, si no cambiábamos nada, no obtenemos el mismo resultado? El problema surgía en la función de split entre train y test de *sklearn*. La documentación nos indica que hay un parámetro *random_state*, que funciona como semilla para poder reproducir ambos sets de igual manera a través de múltiples llamadas. Nosotros no hacíamos uso de esta opción y por ende cada vez que probábamos un modelo se entrenaba con un set distinto de datos. Esto nos provocó que no podamos medir el impacto real que tuvieran las nuevas features que íbamos agregando, obteniendo resultados muy dispares. ¿Entonces habrá algún valor de semilla que genere un set de train y validación, más óptimo que otro? La respuesta es que sí, de hecho, haciendo distintas pruebas llegamos a que el valor de 1881 nos generaba sets de train y test con la mayoría de las edificaciones que tuvieron un nivel 2 y 3 de daño y que eran representativos del set de train original. Este pequeño cambio nos generó un incremento en el score de aproximadamente 0.0050.

Por lo tanto, ¿si se entrenara con todo el set de train original se obtendría mejores resultados? Uno pensaría que sí ya que se tienen más datos, pero al probar la opción de todo el dataset vs. El split generado con seed = 1881, nos encontramos que los resultados eran lo mismo. Interpretamos que esto está atribuido al desbalance que mencionamos en la sección de limpieza de datos.

Un claro ejemplo se puede ver en la siguiente matriz de confusión, donde predominan la cantidad de edificaciones con un nivel de grado 2, seguido por aquellas con un nivel de grado 3.



3.3 Feature selection

Una vez que se tienen todos los atributos en un mismo dataframe y luego de un par de pruebas se ve que no siempre hay que entrenar los modelos con la totalidad de features del dataframe. Viendo que para algunos modelos la selección de los features generaba distintos resultados, se buscó la forma de encontrar la combinación optima de features y eliminar todo el ruido posible.

El modelo usado de referencia (en primeras iteraciones) para esta sección es el de Random Forest debido a su estabilidad para mostrar la importancia de cada feature, y que además fue el primer modelo que probamos. Su funcionamiento radica en que los árboles que crea el modelo toman distintos subconjuntos de features elegidos al azar y arrojan distintos resultados. De esta manera, con cientos o miles de árboles el algoritmo adopta una amplia capacidad predictora de cada atributo.

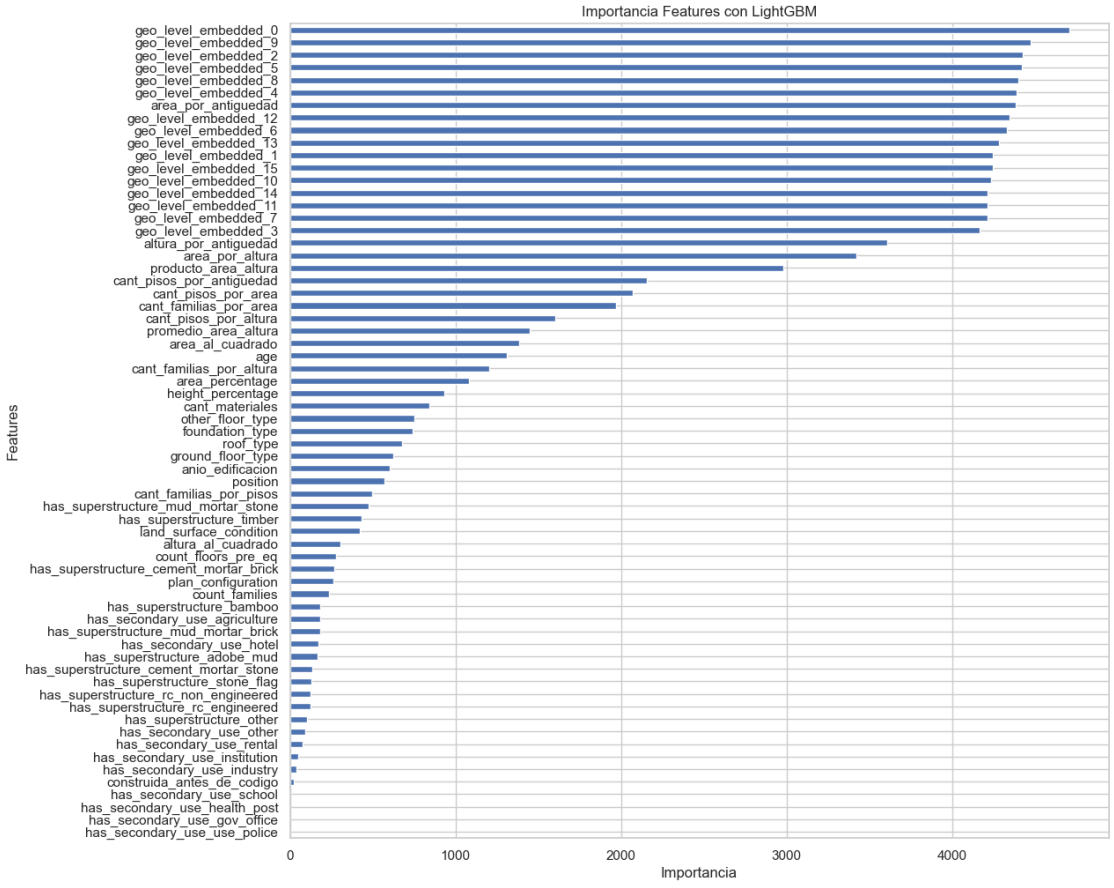
Es importante notar que a mayor cantidad de árboles generados por Random Forest, más precisa será la importancia del feature, ya que más árboles al azar tendrán los features y mejor se podrá estimar cuánto aportaron en cada árbol. Sin embargo, no siempre la importancia que Random Forest le da a un cierto atributo es la misma importancia que le da otro modelo, pero es un buen baseline sobre todo para los modelos basados en árboles. Es por esto por lo que también es prudente interpretar la importancia de los features del modelo con que se está trabajando.

¿Pero entonces como seleccionaron las features? Si bien conocemos varias heurísticas y varias de ellas ya vienen implementadas en *sklearn* como (*backward elimination, forward selection, stepwise selection)*, el proceso de selección se fue haciendo manualmente considerando la importancia de Random Forest y del modelo en cuestión sobre el cual se esté trabajando. Esta forma de trabajar es costosa en tiempo, pues al sacar una o varias features menos importantes debemos entrenar de nuevo el modelo y ver los resultados. A pesar de esto, fue la manera más confiable y segura que encontramos y la que nos dio resultados bastante buenos. También probamos *RecursiveFeatureEliminationCV* de *sklearn*, pero además de tardar muchísimo más que la selección a mano, siempre nos devolvía la totalidad de las features como solución óptima a pesar de que no era cierto pues al probar a mano sacando varias features, obteníamos mejor puntaje.

En cuanto a nuestro dataset, de las 108 nuevas features generadas solo nos quedamos con 13 de ellas, además de las originales que vienen con el dataset. Por lo tanto, en general entrenábamos con 65 features más halla de alguna variación en particular. Esto puede parecer muy greedy ya que este conjunto de features fue probado en modelos basados en arboles (Random Forest, XGBoost, LightGBM) pero como ya habíamos invertido tiempo en la selección, no podíamos permitirnos seguir testeando para cada modelo utilizado cuando todavía nos quedaba la parte de tuneo de hiperparámetros.

Un caso curioso es que para algunos modelos al remover features que tienen una importancia muy cercana a 0, empeoraban, pero no nos brindaba una disminución en el tiempo de entrenamiento. Por lo tanto, se decidió dejar dichas features. Entendemos que esto viene ligado a que es necesario ajustar algunos hiperparámetros una vez removido algún feature en particular para remediar esto.

A continuación, se deja un ejemplo de las 65 features seleccionadas junto con su importancia en uno de los modelos (LightGBM) que mejor performance tuvo.



4. Modelos

4.1 Random Forest

Se constituye de un ensamble de árboles de decisiones con subconjuntos de los atributos a utilizar, y con la técnica de bagging se selecciona finalmente el promedio de los árboles corridos.

Este modelo popular trajo buenos resultados, bastante cercanos a los obtenidos por LightGBM. Una ventaja de este modelo es la capacidad de evitar el overfitting, por lo cual es de los modelos más confiables del presente trabajo. La desventaja es que, a mayor cantidad de árboles, no necesariamente se obtienen mejores resultados, pero el tiempo de entrenamiento se incrementa sustancialmente.

Sus tres hiperparametros importantes son la cantidad de árboles a utilizar, la cantidad de atributos por árbol y la cantidad mínima de muestras para generar un split. Otra característica de este modelo es su invariante frente a la escala de los atributos, es decir, no hay que normalizar los datos.

En cuanto al score, los mejores que obtuvimos fueron:

- Training score: 0.8065
- Testing score: 0.7567
- DrivenData score: 0.7441

4.2 XGBoost

XGBoost es un modelo muy eficiente de gradient boosting basado en árboles. Uno de sus hiperparámetros más importantes es la función objetivo, la cual fue la función *softmax* al tratarse de un problema de clasificación múltiple.

Este modelo fue el siguiente que utilizamos una vez que Random Forest no nos pudo dar mejores resultados con relación al tiempo de entrenamiento.

En cuanto al score, los mejores que obtuvimos fueron:

- Training score: 0.7984
- Testing score: 0.7626
- DrivenData score: 0.7453

4.3 KNN

El primer algoritmo de Machine Learning visto en la materia. Un algoritmo sencillo que consiste en encontrar los vecinos más cercanos del punto a clasificar, y luego simplemente predecir que el punto en cuestión es de la clase del cual la mayoría de sus vecinos sean parte.

Sus hiperparámetros a definir son la cantidad de vecinos a tomar en cuenta y la distancia a utilizar entre ellos.

Paradójicamente, el ser tan sencillo es su máxima desventaja: es un algoritmo de orden cuadrático y esto impacta mucho a la hora de predecir. Notar que no tiene tiempo de entrenamiento, pero a mayor es el dataset peor es el rendimiento de KNN en cuestión de tiempo. En nuestro caso, tardó aproximadamente 40 minutos en predecir el set de train junto al set de validación, y unos 14 minutos en predecir sobre el set de test de DrivenData. Otra cosa para remarcar es que los datos deben estar normalizados a la hora de procesarlos.

Honestamente no esperábamos mucho en cuanto al score, pero los mejores que obtuvimos fueron:

- Training score: 0.7429
- Testing score: 0.7154
- DrivenData score: 0.7095

4.4 Naive-Bayes

Este algoritmo, basado en el teorema de Bayes, es sencillo y bastante rápido tanto al predecir como al entrenar. No suele tener buenos resultados, pero era un algoritmo que debíamos probar al haber cursado la materia Probabilidad y Estadística. Su inocencia proviene del supuesto de que las dimensiones son independientes entre sí. La variante que se probó fue la Gaussiana debido a que nuestro dataset contaba con features continuos (y discretos, pero la variante multinomial no aceptaba valores negativos). Otra cosa para remarcar es que los datos deben estar normalizados a la hora de procesarlos.

En cuanto al score, los mejores que obtuvimos fueron:

- Training score: 0.4244
- Testing score: 0.4247

- DrivenData score: 0.4256

4.5 LightGBM

Nuestro modelo insignia y el que mejor resultado nos dio, tanto en términos de score como tiempo de entrenamiento pensando en el tuneo de hiperparámetros. Este modelo de gradient boosting basado en árboles se diferencia de XGBoost en que construye los árboles según las hojas, y no los niveles. Es sumamente importante que sus hiperparametros estén bien configurados porque es acá donde se le saca todo el provecho. Un LightGBM default da resultados tan malos que te da la impresión de que era mentira todo lo que leíste sobre el modelo.

En cuanto al score, los mejores que obtuvimos fueron:

- Training score: 0.8199
- Testing score: 0.7521
- DrivenData score: 0.7490

4.6 SVM

Un algoritmo lineal que tiene funcionamiento similar a Perceptrón. En Perceptrón se busca algún hiperplano que separe las clases que queremos clasificar; en SVM el objetivo es encontrar el mejor hiperplano que cumpla ese objetivo. Esperábamos mejores resultados de este algoritmo y de la variante *SGDClassifier* de *sklearn* utilizada. De cosas positivas que sacamos es que casi ni overfittea y el tiempo de entrenamiento y predicción es casi nulo.

En cuanto al score, los mejores que obtuvimos fueron:

- Training score: 0.6173
- Testing score: 0.6185
- DrivenData score: 0.6156

4.7 Ensambles y voting a mano

Con el objetivo de intentar obtener un mejor score en la competencia se decidió realizar un ensamble de 5-Fold LightGBM, todos entrenados con los mismo hiperparámetros salvo en la cantidad de árboles a utilizar que iba a ser determinada mediante *cross-validation* y mediante *early stopping*. La idea era simple, como la API de LightGBM nos devolvía las probabilidades para cada clase a predecir por cada edificación, entonces podíamos sumar dichas probabilidades para cada edificación devuelta por cada modelo y quedarnos con la clase que acumule mayor probabilidad. Esta forma de trabajar dista de la otra forma usual de encarar un ensamble, como puede ser mediante votación en donde simplemente nos quedamos con la clase que mayor se repite por cada registro. ¿Pero cómo se diferencia exactamente la votación respecto de la suma de probabilidades? Supongamos que tenemos [[0.05, 0.75, 0.20], [0.10, 0.65, 0.25], [0.15, 0.05, 0.8]] para un determinado registro, en donde cada array es el resultado de la predicción de probabilidades de un modelo. Si tomáramos simplemente la mayor probabilidad entonces el registro pertenecería a la clase 3 pues la mayor probabilidad individual cae en el tercer elemento del tercer array. Sin embargo, si sumamos vemos que la clase 2 suma un total de 1.45 y la clase 3 suma un valor de 1.25 y por lo tanto podemos decir que el mayor intervalo de confianza lo tiene la clase 2 al ser un poco más estable.

Con esta metodología los mejores resultados arrojados fueron:

- Training score: 0.8285
- Testing score: 0.7501
- DrivenData score: 0.7514

Por otro lado, y con la esperanza de mejorar aún mas el score de DrivenData decidimos tomar los 3 submits que mayor resultado nos dieron pero que se hallan entrenado con distintas

features. Como teníamos almacenados los archivos *.csv* que submitteamos fue muy fácil realizar esto, puesto que con solo mergear los resultados y hallar la moda por cada registro obtendríamos el resultado final.

Con esta metodología el mejor resultado que se llegó a obtener fue:

- DrivenData score: 0.7517. El cuál es nuestro mejor resultado final.

5. Parameter tuning

En los primeros modelos corridos fue cuando se empezó a notar lo que ya nos temíamos: los hiperparametros son sencillamente lo más importante de cada algoritmo, y lo que marca la diferencia entre un mismo modelo con buen funcionamiento, con un funcionamiento promedio o con un funcionamiento malo.

Optamos por usar la clase *GridSearchCV* que proporciona *sklearn*. Este proceso (y en particular con GridSearch y con cross-validation) es muy costoso en tiempo, pero nuestra idea fue solamente profundizar los hiperparámetros de aquellos modelos que nos dieran un resultado muy bueno con los parámetros default. Es por esto por lo que se tunearon los hiperparametros de *XGBoost*, *LightGBM* y *Random Forest* basándonos en la documentación oficial de cada modelo y leyendo diversos artículos que se pueden encontrar en internet.

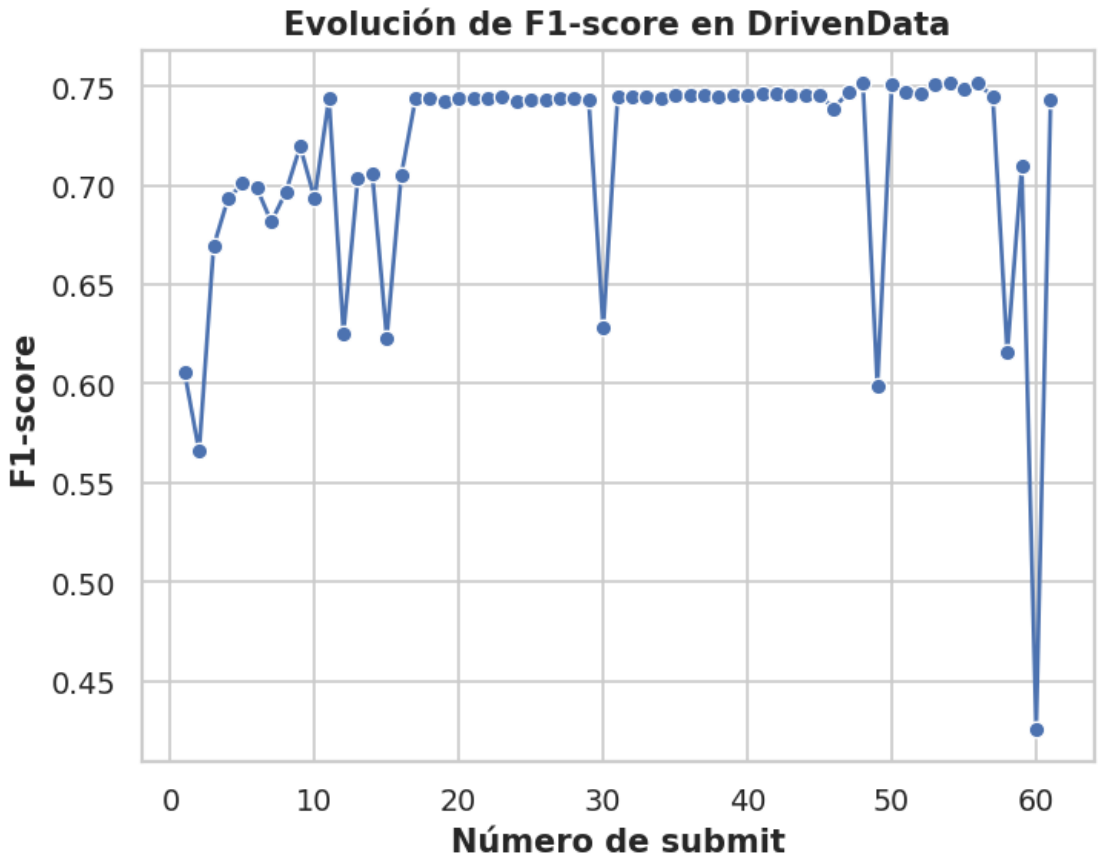
¿Pusieron todos los hiperparametros en una grilla y dejaron correr los algoritmos esperando encontrar la mejor solución? Si y no. Un aspecto importante por entender sobre esta etapa es que no podemos variar a lo loco y sin pensar los hiperparámetros de un modelo. Existen hiperparámetros que están intrínsecamente relacionados con otros y se deben tunear al mismo tiempo. Por ejemplo: en LightGBM es usual disminuir el learning rate siempre y cuando se aumente la cantidad de árboles a construir; también es usual aumentar la cantidad de hojas si se aumenta la profundidad máxima de los árboles, pero al escoger un numero de hojas muy grandes podemos llegar a caer en una situación de overfitting y para combatir esto debemos configurar otros hiperparametros.

6. Tabla de resultados y evolución de score en DrivenData

Como en todo trabajo científico, es necesario una forma de mostrar de manera resumida y numérica todo lo que se habló en el presente trabajo razón por la cual se presenta una tabla que contiene los modelos probados, sus mejores hiperparametros encontrados, los scores en instancias de pruebas locales y de DrivenData; así como también los tiempos de entrenamiento y de predicción de cada uno de ellos (esto último puede variar dependiendo en las condiciones se ejecute, por eso solo ponemos una aproximación).

Modelo	Hiperparametros	Training score	Testing score	DrivenData score	Tiempo de entrenamiento	Tiempo de predicción
Random Forest	'min_samples_split':60, 'n_estimators':1000, 'n_jobs':-1	0.8065	0.7567	0.7441	~ 5 min	< 5 seg
XGBoost	'booster': 'gbtree', 'learning_rate': 0.1, 'n_estimators': 500, 'n_jobs': -1, 'max_depth':6, 'min_child_weight':0.1, 'colsample_bytree':0.6, 'subsample':0.8, 'num_class': 3, 'objective':'multi:softmax'	0.7984	0.7626	0.7453	~ 5 min	< 5 seg
KNN	'n_neighbors': 15, 'n_jobs': -1	0.7429	0.7154	0.7095	-	~15 min
Naive-Bayes	-	0.4244	0.4247	0.4256	~ 2 seg	< 2 seg
LightGBM	'boosting_type': 'gbdt', 'feature_fraction': 0.6, 'learning_rate': 0.1, 'max_bin': 7000, 'max_depth':-1, 'metric': 'multi_logloss', 'min_data_in_leaf': 120, 'min_sum_hessian_in_leaf': 0.1, 'n_jobs': -1, 'n_estimators':929, 'num_class': 3, 'num_leaves': 40, 'objective': 'multiclass', 'random_state': 1881,	0.8199	0.7521	0.7490	~ 2 min	< 5 seg
SVM	'loss': 'hinge', 'penalty': 'l2', 'alpha':0.0001, 'max_iter':1000, 'early_stopping':True	0.6173	0.6185	0.6156	< 2 seg	< 2 seg
Ensamble (5-Fold LGBM)	Mismos que LGBM, pero con número de árboles variable por cada instancia de LGBM	0.8285	0.7501	0.7514	~ 30 min	< 10 seg
Voting a mano	-	-	-	0.7517	-	-

Por otra parte, presentamos el progreso de nuestro F1-score en DrivenData a lo largo de los distintos submits que fuimos realizando.



Cada uno de los puntos fueron producidos por distintos factores, entre los que se encuentran:

- Submits correspondientes a distintos modelos.
- Para un determinado modelo, se probaron distinta configuración de hiperparámetros.
- Para un set de hiperparámetros fijos para un modelo, se fue variando la cantidad de features a utilizar.

7. Conclusiones

- **¿Qué aprendimos?** En primer lugar, pudimos comprobar empíricamente que Machine Learning requiere de una serie de pasos ordenados para trabajar. Pudimos observar que con el agregado de features que a nuestra visión pudieran ser irrelevantes para una predicción, el score mejoró significativamente. Además, la sección de feature engineering es mucho más importante (a nuestro parecer) que la de tuneo de hiperparametros ¿Por qué? El agregado de features, así como también los distintos tipos de encoding que se puedan aplicar, producen resultados casi inmediatos en la métrica a utilizar; mientras que el tuneo de hiperparámetros requiere muchísimo tiempo y mejora los resultados de un modelo, se suele llegar a un limite en donde por más que se siga invirtiendo tiempo en hallar los hiperparámetros óptimos, la mejora no es significativa. Por esto, creemos que hay que ser creativos y preguntarse como minar cada una de las features iniciales para obtener el mayor resultado posible. De la mano de lo recién mencionado, es importante cómo se seleccionan las features para un modelo. Por ejemplo: una feature por si sola puede no aportar nada a un modelo, pero tal vez con la interacción de alguna otra puede producir cambios positivos.

- **¿Qué fue lo más interesante sobre la competencia?** Para este set de datos en particular, lo mas interesante fue hallar los elementos mas determinantes a la hora de determinar el nivel de daño de una propiedad. Antes de empezar con el presente trabajo práctico, creíamos que el tipo de estructura junto con algunas características intrínsecas de la edificación podrían tener mayor incidencia en los modelos, y nos encontramos que con solo saber la ubicación de una vivienda podemos tener un grado de confianza importante al momento de predecir el grado de daño. A nivel modelos de Machine Learning, nos sorprendió LightGBM no solo por su velocidad a la hora de entrenar si no por la precisión en las predicciones que nos brindaba.
- **¿Qué nos quedó por mejorar?** Algo que nos faltó fue encontrar alguna forma de trabajar o minar las columnas booleanas del set de datos original. Podríamos haber probado algunos algoritmos de reducción de dimensiones que se vieron en la materia y comparar resultados con/sin estos algoritmos. También nos hubiera gustado probar otros tipos de encoding, así como también tunear el hiperparámetro de target encoding utilizado. Al final de la competencia nos dimos cuenta de que estábamos filtrando el label a nuestro set de validación debido a que primero aplicábamos el encoding y luego splitteabamos el set de datos. Por último, podríamos haber implementado alguna red neuronal por más que no sea el estado del arte para datos estructurados.