FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TRABAJO PRÁCTICO 2

MACHINE LEARNING

[75.06 / 95.58] ORGANIZACIÓN DE DATOS

SEGUNDO CUATRIMESTRE DE 2019

INTEGRANTES

Apellido	Nombre	Padrón	Mail
Aguerre	Nicolás	102145	naguerre@fi.uba.ar
Parafati	Mauro	102749	mparafati@fi.uba.ar
Secchi	Ana María	99131	asecchi@fi.uba.ar

ÍNDICE

LINKS PRÁCTICOS

- Repositorio (GitHub): github.com/nicomatex/datos_tp2_2c2019.

1 INTRODUCCIÓN

En este informe se busca documentar y explicar el trabajo realizado a lo largo de la segunda parte de la materia. El proyecto consistió en intentar predecir los precios de las propiedades publicadas en **ZonaProp (México)** a partir del análisis de sus atributos. Para lograr el objetivo, se trabajó con distintos modelos y algoritmos de **Machine Learning** que serán explicados y detallados en el presente informe.

Por tratarse de un trabajo sumamente extenso en el que se probaron muchas cosas, primero explicaremos en detalle como nos organizamos, para luego poder detallar, sección por sección, todas las ideas analizadas y alternativas tomadas para lograr el modelo final.

Finalmente desarrollaremos una serie de conclusiones o *insights* que resulten de interés para el lector.

2 ORGANIZACIÓN

En un proyecto de tal magnitud como es un trabajo de Machine Learning, resulta indispensable la organización del mismo, no sólo por el hecho de que hay muchas cosas para hacer, sino porque es un trabajo que se realiza en equipo y es necesario que las tareas se puedan independizar para así lograr resolver el problema de manera eficaz.

2.1 Pipeline propuesto

En un primer paso, nos propusimos armar una estructura de tipo **pipeline***, identificando las distintas etapas del trabajo, independizandolas y automatizandolas, para así permitir el trabajo en paralelo entre los distintos miembros del equipo. Definimos entonces, las siguientes etapas:

- **Análisis exploratorio**: primer etapa que se encontró cubierta por el **Trabajo Práctico 1** realizado anteriormente. La idea es intentar entender con qué datos estamos tratando, que información tenemos y que no tenemos, y empezar a visualizar distintas relaciones entre las variables que nos sean de ayuda a la hora de proponer modelos.
- **Pre-procesamiento**: en esta segunda etapa, la tarea será realizar un procesamiento de los datos a fines de obtener distintos beneficios, como sean ahorrar memoria, facilitar el manejo de los dataframes, y realizar tareas previas tales como preparación de datos externos, de ser necesario.
- **Feature Engineering**: tercer etapa y probablemente la más importante. Si bien contamos con unos pocos atributos de cada propiedad, es casi seguro que haya datos ocultos en los mismos. Este proceso se basa en intentar extraer la mayor información posible a las variables iniciales, para tener más datos con que "alimentar" a nuestros modelos.

  Por tratarse de un proceso tan importante y largo, decidimos subdividirlo en dos: **Feature Generation**, donde partiendo de los atributos iniciales generaremos nuevas features, y **Feature Selection**, donde una vez generados los features (posiblemente un largo numero) se probarán distintos algoritmos y distintas técnicas para quedarnos con los que más aporten al modelo, y para intentar eliminar ruido (hablaremos de esto más adelante).
- **Modelos**: una vez obtenidos los features con los que entrenaremos, es hora de elegir los modelos a probar. La idea es encontrar rápidamente dos o tres modelos que nos generen buenos resultados para luego profundizar en la

optimización de los mismos, porque si bien existen muchos modelos, sabemos que el modelo óptimo sólo se conseguira tras trabajar mucho sobre un modelo inicial.

- **Parameter Tuning**: una vez definidos los features, y el modelo a utilizar, sólo queda optimizar el mismo para que logre los mejores resultados posibles, intentando evitar el **Over Fitting**, y buscando hiper parámetros para lograr mejores resultados. Estos se buscarán utilizando distintos algoritmos, así como probando manualmente.
- **Submission**: por último, entrenaremos el modelo y predeciremos los precios para las propiedades objetivo: compararemos distintos métodos y probaremos interacción entre los mismos, a fines de quedarnos con las predicciones más precisas.

Para facilitar el entendimiento del lector, intentaremos visualizar de forma gráfica la propuesta:

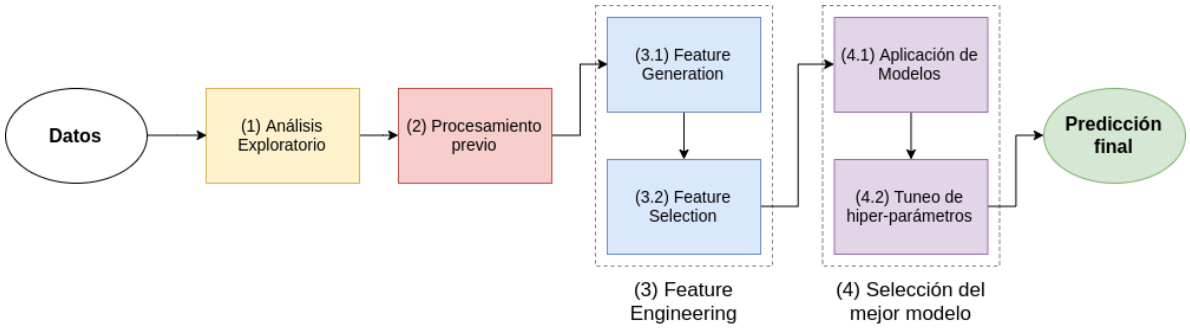

Figura 1: Organización del trabajo

* *"El pipeline es una técnica para implementar el paralelismo a nivel de instrucciones dentro de un solo procesador. Pipelining intenta mantener ocupada a cada parte del procesador, dividiendo las instrucciones entrantes en una serie de pasos secuenciales, que se realizan por diferentes unidades del procesador que trabajan de forma simultánea."*

3 ANÁLISIS EXPLORATORIO

En todo trabajo de Machine Learning como primer paso resulta imprescindible entender **con qué datos estamos tratando**, para así poder orientar nuestro procesamiento en la dirección correcta. Este análisis es una parte muy extensa de cualquier proyecto, por lo que fue realizado aparte como otro trabajo, en el **Trabajo Práctico 1**.

Como nos resultará util recordar las conclusiones encontradas, detallamos a continuación la información más relevante que obtuvimos como consecuencia del desarrollo del mismo:

- **Distribución del tipo de propiedad**: se observó que aproximádamente un 70 % de los datos son **Casas** y otro 25 % **Apartamentos**.
- **Distribución de atributos extra**: se observó que aproximadamente un 90 % de las propiedades poseen **garaje**, un 45 % poseen **Escuelas Cercanas**, y sólo un 10 % poseen **Piscina**.
- **Datos faltantes**: pudimos ver que en los campos de **latitud** y **longitud**, sólo el 50 % de las propiedades cuentan con información, mientras que en otros campos importantes como **metros totales** y **metros cubiertos**, tenemos datos inconsistentes o sin sentido en más de 80,000 oportunidades, lo que representa aproximádamente un 25 % de los datos.
- **Distribución de otras variables**: aproximadamente un 25 % de las propiedades están **a estrenar**, mientras que casi el 60 % tienen una antigüedad **menor a 10 años**. Mas de la mitad de las propiedades tienen **tres habitaciones** mientras que otro 25 % tienen **dos habitaciones**.
- **Distribución de provincias**: descubrimos que la mayoría de las propiedades publicadas pertenecen a **Distrito Federal**, cuyo precio aumenta notablemente respecto del resto de las provincias.
- **Anomalías en la fecha de publicación**: se observó que en **Diciembre del 2016** se publicaron muchas más propiedades que en cualquier otro mes. Esto nos llevó a pensar que esas fechas podrían ser erróneas en los datos.
- **Correlación lineal entre los metros cubiertos y el precio**: con mucho sentido, observamos una relación lineal entre los metros y el precio.
- **Los precios crecientes en el tiempo**: se observó que los precios aumentan con el pasar de los años, de donde se dedujo que una posible causa era la devaluación de la moneda mexicana frente al dólar, y la inflación local.

Consideramos de gran importancia haber realizado este análisis previo, ya que permite introducirse en los datos y detectar posibles fuentes de error.

4 PROCESAMIENTO PREVIO DE LOS DATOS

En esta etapa se busco realizar un tratamiento sobre los datos para facilitar su posterior manejo, y para optimizar el manejo de recursos. En los notebooks **pre_processing** y **data_purging** se definieron distintas funciones y se probaron distintos algoritmos. A su vez, se procesaron datasets externos que fueron introducidos al problema.

4.1 Dataset externo: cotización MXN-USD

Motivación: como vimos en el TP1, los precios de las propiedades publicadas crecían de forma monótona con el pasar de los años. Esto nos llevo a pensar en que tal vez, una posible causa de este aumento era el **aumento de la cotización del peso mexicano frente a la moneda norteamericana**. Para darle esta información al modelo, decidimos incorporar un dataset con la información actualizada en el período de fechas de interes.

4.2 Dataset externo: ubicación de volcanes con riesgo de activación

Motivación: es lógico creer que si una propiedad posee riesgos naturales cerca, el precio de la misma debería disminuir. No pudimos obtener ningún dataset gratuito que nos brinde información sobre otros desastres naturales, pero sí pudimos obtener este que nos ofrece la ubicación de los distintos volcanes con riesgo de activación en el territorio mexicano.

4.3 Deteccion de Outliers

Motivacion: Se observo que habian propiedades cuyo precio no tenia sentido (en relacion a sus atributos, como superficie y antiguedad). De esta observacion surgio la sospecha de que el dataset de entrenamiento podia contener una cierta cantidad de outliers que estuvieran afectando las predicciones del modelo que se estaba entrenando.

4.3.1 *Con DBScan*

DBScan es en realidad un algoritmo de Clustering basado en densidad, que consiste en clasificar la totalidad de los puntos (que representan a los datos) o bien como parte de un cierto cluster, o bien como parte de un ruido. Este algoritmo tiene dos hiperparametros principales: una cierta distancia e que es la distancia maxima para la cual dos puntos se consideran vecinos, y una cierta cantidad n de puntos minimos para que una region se considere densa. Al final de este algoritmo, todos los puntos que que no pertenecen a ningun cluster son considerados Outliers.
La aplicacion de este algoritmo desafortunadamente no condujo a resultados visiblemente mejores, a pesar de la sospecha de que el dataset contiene una buena cantidad de outliers.

4.4 Isolation Forest para la detección de outliers

Isolation Forest es un algoritmo que, al contrario que DBScan, se basa en aislar los outliers, en lugar de clusterizar los puntos normales. En escencia, se elige al azar una cierta feature y un cierto valor (entre el minimo y el maximo valor de esta feature) y se realiza una particion de los datos a partir de este valor. Este proceso se realiza de forma recursiva para features aleatorias, hasta que queda una cierta particion queda con un unico punto *aislado*. Este punto es entonces clasificado como outlier.

La cantidad de outliers detectados con este algoritmo no fue demasiado diferente respecto a DBScan, y el resultado tampoco. No hubo una mejoria visible en los resultados del modelo luego de entrenarlo usando un set de entrenamiento sin los outliers hallados por este algoritmo.

4.5 Imputación de nulos para el campo tipodepropiedad

Motivación: se observó que habían unas pocas publicaciones que no tenían ningún valor en el campo **tipodepropiedad**, pero que se podía deducir el valor del mismo a partir de otros campos, como la **descripción** o el **título**. Se procesó el texto para finalmente llenar estos nulos.

5 FEATURE GENERATION

Consideramos que esta es la etapa más importante en cualquier proyecto de Machine Learning. Las **features** son esenciales, ya que son las que brindan al modelo la información para que este pueda predecir el **target**, en este caso, el precio de las propiedades. Inicialmente, contamos con unas pocas features que se listan a continuación:

```
{'titulo', 'descripcion', 'tipodepropiedad', 'direccion', 'ciudad',
'provincia', 'antiguedad', 'habitaciones', 'garages', 'banos',
'metroscubiertos', 'metrostotales', 'idzona', 'lat', 'lng',
'fecha', 'gimnasio', 'usosmultiples', 'piscina', 'escuelascercanas',
'centroscomercialescercanos'}
```

Y vemos que muchas de estas deben ser tratadas especialmente, por ejemplo, las categóricas (los modelos aceptan sólo variables numéricas), la fecha (buscar la manera óptima de representarla), los títulos y descripciones (¿se puede sacar información extra?), etc.

A lo largo de esta sección, trabajaremos sobre estas variables iniciales para terminar con muchas más, y obtener resultados más precisos. Por cuestiones de organización, se decidió seguir un protocolo de generación de features que facilite su posterior selección: generamos un diccionario con distintos grupos de features en el que, para cada grupo, almacenamos los nombres de las features generadas, para luego poder buscar las mejores features para cada grupo, y posteriormente incorporarlas al modelo.

Para facilitar la compresión al lector, se propone un ejemplo: en el caso de **tipodepropiedad**, se probarán muchos métodos de encoding, pero la información será la misma. Se almacenan entonces, en este diccionario, todas las alternativas que se probaron para luego seleccionar cual fue la mejor.

5.1 Encoding de variables categóricas

Primero trabajaremos sobre las variables categóricas, ya que como explicamos, los modelos que utilizaremos no admiten variables de este tipo (sólo admiten variables numéricas). Las variables categóricas con las que trabajaremos en esta sección son:

```
{'tipodepropiedad', 'ciudad', 'provincia'}
```

Probaremos distintas formas de "encodear" estas variables para transformarlas en un tipo de dato que nuestro modelo acepte.

5.1.1 *Encoding de tipodepropiedad*

Primero, como sabemos que las clases minoritarias pueden ser problemáticas y generar overfitting en el modelo, agrupamos los tipos de propieades similares en un mismo grupo (por ejemplo, *terreno* y *terreno industrial*), para quedarnos con menos clases y con mayor muestra de cada una de estas.

Una vez realizado este proceso, generamos nuevas features utilizando distintos métodos de encoding:

- **Label Encoding**: método básico que consiste en asignarle un número a cada clase. Como desventaja, se puede introducir una ordinalidad en la información que no tenga sentido y confunda al modelo. Se generó entonces la siguiente feature:

  ```
  {'tipodepropiedad_le'}
  ```

- **One Hot Encoding**: método muy eficiente que se utiliza muchísimo. La idea es generar tantas columnas como clases se tengan, y estas columnas tomaran valores binarios, que serán 1 en el caso de que el dato pertenezca a dicha clase. One Hot Encoding es muy eficiente cuando se trata con una cardinalidad razonablemente baja. Cuando el número de clases aumenta, puede ser mejor utilizar label encoding o otro método. Se generaron las siguientes features:

```
{'tipodepropiedad_1_ohe', 'tipodepropiedad_2_ohe', ...
'tipodepropiedad_7_ohe', 'tipodepropiedad_8_ohe'}
```

- **Binary Encoding**: método que calcula cuantos dígitos son necesarios para representar a las clases en binario, y luego genera dicha cantidad de columnas para asignarles su valor binario. Se generaron las siguientes features:

```
{'tipodepropiedad_0_binary', 'tipodepropiedad_1_binary',
'tipodepropiedad_2_binary', 'tipodepropiedad_3_binary'}
```

- **Polynomial Encoding**: otro método de encoding menos conocido que nos brinda sklearn, generó las siguientes features:

```
{'intercept_pol', 'tipodepropiedad_0_pol', 'tipodepropiedad_1_pol', ...
'tipodepropiedad_5_pol', 'tipodepropiedad_6_pol'}
```

- **Mean Encoding**: método también muy usado, sobre todo cuando la cardinalidad es muy alta. Se basa en, para cada clase, reemplazarla por un promedio del target para dicha clase. Este método debe ser utilizado con mucho cuidado, ya que es muy difícil que no genere overfitting (existen algunas modificaciones para prevernilo). Generamos la siguiente feature:

```
{'tipodepropiedad_me_m0'}
```

- **Mean Encoding + Smoothing**: nuevamente mean encoding, pero esta vez incorporando una técnica para prevenir overfitting que le da peso también al promedio general, y no sólo al de la clase. Se generaron varias features, con distintos números de m (m es el peso que se le da al promedio general):

```
{'tipodepropiedad_me_m1', 'tipodepropiedad_me_m2',
'tipodepropiedad_me_m3', 'tipodepropiedad_me_m4'}
```

- **Encoding manual**: en vistas de lo observado en el TP1, notamos que es de vital importancia determinar si se trata de una Casa o un Apartamento, por lo que generamos las siguientes features para identificarlas:

```
{'es_casa', 'es_apart'}
```

- **Alternativas con malos resultados**: se probó también utilizar **Expanding Mean**, que es otra variante del mean encoding utilizado, pero esta tuvo resultados muy malos, por lo que se decidió no incorporarla al modelo.

5.1.2 *Encoding de provincia*

Se siguió el mismo procedimiento que con tipodepropiedad, y se utilizaron los mismos métodos de encoding. Se generaron features utilizando **Label Encoding**, **One Hot Encoding**, **Binary Encoding**, **Mean Encoding**, **Mean Encoding + Smoothing**, y **Encoding manual** (se armó un top 10 de las provincias con mayor muestra de propiedades, y se las encodeó utilizando OHE). Features generadas:

```
{'provincia_le'}
{'provincia_1_ohe', 'provincia_2_ohe', ... 'provincia_33_ohe'}
{'provincia_0_binary', 'provincia_1_binary', ... 'provincia_6_binary'}
{'es_Distrito Federal', 'es_Edo. de México', 'es_Jalisco', 'es_Querétaro',
'es_Nuevo León', 'es_Puebla', 'es_San luis Potosí', 'es_Yucatán',
'es_Morelos', 'es_Veracruz'}
{'provincia_me_m0'}
{'provincia_me_m1', 'provincia_me_m2', 'provincia_me_m3', 'provincia_me_m4'}
```

5.1.3 *Encoding de ciudad*

Nuevamente, utilizamos los mismos métodos de encoding. Esta variable, sin embargo, hay que tratarla con más cuidado: tiene más de 900 clases, por lo que realizar one hot encoding resultaría una locura sin antes agrupar. De todas formas, probamos con los siguientes métodos: **Label Encoding**, **One Hot Encoding** (quedandonos con las 50 ciudades con mayor muestra, y agrupando al resto en otra clase), **Binary Encoding**, **Mean Encoding**, **Mean Encoding + Smoothing**, y **Encoding manual** (generamos tres features nuevas analizando el precio promedio para cada ciudad que reflejan si se trata de una ciudad cara, barata, o normal). Features generadas:

```
{'ciudad_le'}
{'ciudad_top50_1_ohe', 'ciudad_top50_2_ohe', ... 'ciudad_top50_50_ohe'}
{'ciudad_0_binary', 'ciudad_1_binary', ... 'ciudad_7_binary'}
{'ciudad_cara', 'ciudad_barata', 'ciudad_normal'}
{'ciudad_me_m0'}
{'ciudad_me_m1', 'ciudad_me_m2', 'ciudad_me_m3', 'ciudad_me_m4'}
```

5.1.4 *Binning y encoding de otras variables*

Si bien las tres anteriores variables analizadas eran las únicas variables categóricas originalmente, utilizamos el concepto de **binning** para generar nuevas variables categóricas a partir de variables numéricas, con el objetivo de ver si esto aumentaba la precisión del modelo. Se realizó este procedimiento para las variables **antigüedad** (utilizando dos funciones de binning distintas) y **habitaciones**, para luego encodear los resultados utilizando los métodos vistos recién, generando las siguientes features:

```
{'antiguedad_binning_1_ohe1', ... 'antiguedad_binning_9_ohe1'}
{'antiguedad_me_m0'}
{'antiguedad_me_m1', 'antiguedad_me_m2', 'antiguedad_me_m3', 'antiguedad_me_m4'}
{'antiguedad_binning_2_1_ohe2', 'antiguedad_binning_2_2_ohe2',
'antiguedad_binning_2_3_ohe2', 'antiguedad_binning_2_4_ohe2'}
{'antiguedad2_me_m0'}
{'antiguedad2_me_m1', 'antiguedad2_me_m2', 'antiguedad2_me_m3', 'antiguedad2_me_m4'}

{'hab_binning_1_ohe', ... 'hab_binning_7_ohe'}
{'hab_binning_me_m0'}
{'hab_binning_me_m1', 'hab_binning_me_m2', 'hab_binning_me_m3',
'hab_binning_me_m4'}
```

5.1.5 *Encoding de ID Zona*

Si bien el modelo acepta esta variable tal como viene, vamos a aplicarle **Mean Encoding**, teniendo en cuenta que es muy probable que los barrios tengan un gran impacto sobre el precio de la propiedad. Generamos las siguientes features, agregando smoothing:

```
{'idzona_meanencoding_m0'}
```

```
{'idzona_meanencoding_m1', 'idzona_meanencoding_m2',
'idzona_meanencoding_m3', 'idzona_meanencoding_m4'}
```

5.2 Tratando valores nulos en metrostotales y metroscubiertos

La variable **metrostotales** es muy importante, asi como **metroscubiertos**. Es la principal feature que determina el precio de una propiedad. Lamentablemente, muchas propiedades no tienen datos en estas filas o tienen datos inconsistentes. Para solucionar este problema, proponemos algunas alternativas:

- **Metros confiables**: generamos una columna que indicará con un 1 si los datos tienen sentido (son distintos de nulos, y los metros totales son mayores a los metros cubiertos), y un 0 caso contrario.
- **Utilizar los datos que tenemos**: en el caso en que tengamos datos sobre metrostotales pero no sobre metroscubiertos, o viceversa, una alternativa puede ser duplicar este dato sobre la variable faltante.
- **Metros cubiertos mayores a metros totales**: una posible solución a esta inconsistencia en los datos, resulta invertir los valores.
- **Otra alternativa a metros cubiertos mayores a metros totales**: otra idea puede ser considerar que los metros totales son en realidad, metros descubiertos. Entonces, el valor de metrostotales se calcula como el valor de metrosdescubiertos + metroscubiertos.

Features generados:

```
{'metrostotales_confiables, metrostotales_confiables_alt'}
{'metroscubiertos_alt', 'metrostotales_alt'}
{'metroscubiertos_i1', 'metrostotales_i1'}
{'metroscubiertos_alt', 'metrostotales_i2'}
```

5.3 Manejo de la variable fecha

En todo modelo es muy importante el tratamiento que se le da a la variable temporal, en este caso, **fecha**. Vamos a proponer algunas alternativas para representar esta información de manera eficiente.

Primero generamos algunas features básicas: **anio**, **mes**, **dia**, **timestamp**, **aniomes** (variable numérica que resulta de la concatenación del año con el mes, ej: 201612).

Luego, generamos algunas features del estilo *mean encoding*, nuevamente resaltando que hay que tener cuidado con este tipo de features por la tendencia al overfitting que pueden generar. Estas features fueron: **precio_promedio_metrocubierto_aniomes** y **precio_promedio_metrocubierto_mes**.

Ahora, proponemos corregir algunos problemas que pueden confundir al modelo, como es el espacio entre los valores circulares. En el caso de los meses, por ejemplo, la distancia entre el 12 y el 1 debería ser la misma que entre el 3 y el 4, y esto no sucede en el encoding utilizado actualmente. Para solucionarlo, aplicamos la transformación coseno a las variables **mes**, y **dia**.

También **escalamos entre 0 y 1** otras variables, y **equidistanciamos** otras más. Utilizamos un encoding manual para indicar si la publicación es de **Diciembre del 2016**, dado que gracias al TP1 sabemos que hay un desbalanceo en este mes. También utilizamos **Mean Encoding** para la variable aniomes.

Finalmente, las variables generadas resultaron las siguientes:

```
{'timestamp'}
{'timestamp_scaled'}
{'aniomes'}
```

```
{'precio_promedio_metrocubierto_aniomes', 'precio_promedio_metrocubierto_mes'}
{'anio_featured', 'mes_featured', 'dia_featured'}
{'dic2016'}
{'anio', 'mes'}
{'aniomes_me_m0'}
{'aniomes_me_m1', 'aniomes_me_m2', 'aniomes_me_m3', 'aniomes_me_m4'}
```

5.4 Buscando información extra de los textos

Tenemos dos variables de *texto libre* para el usuario: **titulo** y **descripcion**. Para utilizar estas variables, vamos a intentar extraer información extra que no tenemos inicialmente, como por ejemplo, si la propiedad cuenta con jardín, o terraza. Los indicadores buscados fueron los siguientes:

- **Jardín**: si la propiedad cuenta con jardín.
- **Vigilancia**: si la propiedad cuenta con vigilancia o seguridad.
- **Aire acondicionado**: si la propiedad cuenta con aire acondicionado.
- **Calefacción**: si la propiedad cuenta con calefacción.
- **Ventilador**: si la propiedad cuenta con ventiladores.
- **Parrilla**: si la propiedad cuenta con parrilla.
- **Terraza**: si la propiedad cuenta con terraza.
- **Lujo**: si la propiedad cuenta con jacuzzi o sauna.
- **Cuarto de servicio**: si la propiedad cuenta con cuarto de servicio.

Se generaron, a partir de estos indicadores, las siguientes features:

```
{'jardin', 'vigilancia', 'aire_acondicionado', 'ventilador', 'calefaccion',
'parrilla', 'terraza', 'lujo', 'servicio'}
```

5.5 Algunas métricas de las propiedades

Puede resultar útil para la predicción y el entrenamiento del modelo, conocer cuál es el promedio y el desvío de metros cubiertos por ciudad y por tipo de propiedad. También puede servir tener el precio promedio del metro cuadrado por tipo de propiedad. Generamos las siguientes features:

```
{'metroscubiertos_city_mean', 'metroscubiertos_city_std', 'metroscubiertos_tipo_mean',
'metroscubiertos_tipo_std', mean_city_sqrm_price}
```

5.6 Cotización del peso mexicano frente al dólar

Como explicamos anteriormente, creemos que esta puede ser información interesante que ayude al modelo a entender el aumento de los precios. Generamos las siguientes features:

```
{'usd_variacion', 'usd_precio_promedio_mensual',
'usd_variacion_mes_anterior', 'usd_subio'}
```

5.7 Volcanes con riesgo de activación cercanos

Incorporamos un dataset externo para poder indicar en una nueva feature si la publicación posee o no volcanes cercanos con riesgo de activación. Se generaron las siguientes features:

```
{'volcan_cerca', 'volcanes_cerca'}
```

5.8 Otras features generadas

En esta sección agrupamos aquellas features generadas que no tienen una categoría específica: la feature **apto chicos**, que indica si se trata de una propiedad con mas de 2 habitaciones y con escuelas cercanas, y otra feature **full**, que indica si se trata de una propiedad que posee todas las comodidades: gimnasio, pileta, sum, escuelas cercanas, comercios, etc.

```
{'apto_chicos', 'full'}
```

6 FEATURE SELECTION

Vimos en la sección anterior que hemos generado muchísimas features: más de 300. Sin embargo, sería un grave error intentar entrenar un modelo utilizando estas 300 features y a continuación se detalla porqué:

- **Información repetida**: como explicamos en la parte de feature generation, hemos generado para una misma variable, distintas formas de representar la misma información. Es preciso primero encontrar con cuál de estas formas de representación quedarnos, para no repetir información.
- **Lentitud de los modelos**: mientras más features, más complejo el modelo y más lento se vuelve el proceso del train y el predict. Lo ideal es mantener los features por abajo de los 100, aunque siempre depende de la cantidad de datos que se manejen.
- **Ruido**: contra lo que se podría creer, más features no siempre es mejor. Es muy probable que muchas features sólo introduzcan ruido, confundiendo al modelo y empeorando su performance.

Pero entonces, **¿con qué features nos quedamos?** La realidad es que es practicamente imposible conocer cuál es la combinación de features que mejor resultado da para un determinado modelo, ya que no sólo depende mucho de la métrica utilizada y del azar en la formación de los conjuntos de validación, sino que llevaría **muchísimo tiempo** probar todas las combinaciones posibles.

Por ejemplo, si tuviesemos 10 features, podríamos probar lo siguiente: primero, probar utilizando 1 feature, y tendríamos para elegir 10 features. Luego probamos utilizando 2 features, y tenemos para elegir todas las combinaciones de 2 features de nuestros 10. Y así sucesivamente. Este número se puede calcular (se deduce del polinomio de Newton) y resulta ser 2^n, siendo n el número de features. Es decir, que como en nuestro caso tenemos 300 features aproximádamente, probar con todas las posibles combinaciones sería equivalente a entrenar 2^{300} modelos.

Hecha esta breve introducción al problema, planteamos las alternativas que fueron probadas, en órden cronológico de aplicación.

6.1 Encontrando la mejor alternativa en cada grupo

Como explicamos ya repetidas veces, decidimos definir grupos de features que tengan la misma información representada de distinta forma, con el objetivo de encontrar cuál es la mejor representación para ésta.

Ahora, como recién explicamos, probar todas las combinaciones es imposible, pero si asumimos independencia (sabemos que las features no son independientes, pero dada la separación que realizamos, la independencia entre grupos es más factible), podemos encontrar utilizando una estrategia *greedy*, cuál es la mejor combinación de features para cada categoría. **¿Por qué?** Podemos hacerlo ya que en cada grupo, n es como mucho 10, y $2^{10} = 1024$ combinaciones. Entrenar 1024 modelos puede parecer muy costoso, pero hay modelos para todo. En nuestro caso, utilizando el modelo **LightGBM** con hiper-parámetros apuntando a la velocidad, pudimos llevar a cabo esta estrategia, obteniendo resultados que mejoraron por mucho la precisión de nuestros modelos.

6.1.1 *Procedimiento*

Para cada grupo de features, se probaron todas las combinaciones posibles, guardando las tres mejores en un diccionario. Finalmente, se tomó una decisión humana entre las tres mejores combinaciones, teniendo en cuenta el número de features y qué tipo de features son (si son propensas al overfitting o no, por ejemplo).

6.1.2 *Resultados*

Gracias a este primer procedimiento, logramos reducir el número de features de más de 300 a sólo 150 features aproximadamente, logrando muy buenos resultados.

Por dar algunos ejemplos, observamos lo siguiente:

- En cuanto al encoding de **tipodepropiedad**, el mejor método de encoding resultó ser una combinación entre **Binary Encoding** y **Polynomial Encoding**.
- Para el atributo **provincia**, la mejor combinación resultó de juntar los métodos de **encoding manual** con **Mean Encoding**.
- En cuanto a la variable **ciudad**, se observaron los mejores resultados juntando **One Hot Encoding**, **Mean Encoding** y **métodos de encoding manual**.
- Para la **fecha**, la mejor representación se logró utilizando **Mean Encoding** para el *aniomes* y utilizando también una feature que indica si se trata o no de **Diciembre del 2016** (encoding manual).

6.2 Cumulative Importance

Sobre las features obtenidas con el algoritmo greedy anteriormente explicado, vamos a correr ahora otro algoritmo de naturaleza *greedy* que consiste en el siguiente procedimiento:

1. Se define un modelo a utilizar. Para realizar este proceso, elegimos utilizar **RandomForest** por su conocida estabilidad a la hora de conocer la importancia de los features en un modelo.
2. Una vez definido el modelo, se ordenan de mayor a menor las features según su importancia (*feature importance*).
3. Se genera una lista progresiva agregando las features de a una a la vez. Ejemplo: [['feature1'], ['feature1', 'feature2'], ... ['feature1', 'feature2', ... 'featureN']].
4. Se corre progresivamente el modelo para estas combinaciones de features. Es decir, hay que correr n veces el modelo.

Como cualquier algoritmo de naturaleza greedy, se trata de un algoritmo muy costoso, pero que en muchos casos puede tener resultados muy útiles.

6.2.1 *Resultados*

En nuestro caso, los resultados obtenidos no nos fueron útiles para mejorar la precisión que estabamos logrando ya de por sí en nuestro modelo, pero sí fue útil para entrenar de manera más eficiente los modelos de **RandomForest**. Probablemente si hubiesemos utilizado el modelo final para realizar el procedimiento se hubiesen obtenido resultados aplicables, por lo que es algo a probar en el futuro si se desea mejorar las predicciones.

7 ANÁLISIS DE LA IMPORTANCIA DE LAS FEATURES

Tras haber realizado la selección de features con los métodos descriptos en esta sección, procedemos a analizar de forma gráfica cuál fue la importancia que le dieron los modelos que más utilizamos a nuestras features.

7.1 LightGBM

Se grafica a continuación la importancia de las 25 features más importantes según **LightGBM**:

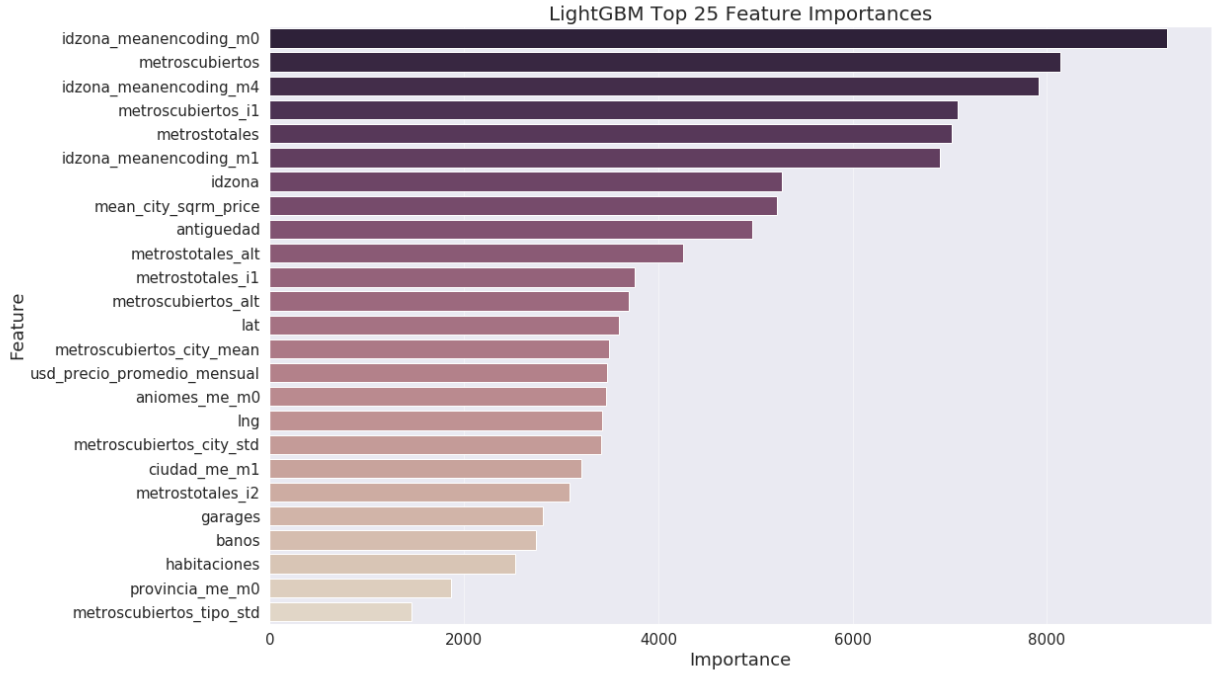

Figura 2: Feature Importance para LightGBM

En este primer gráfico, vemos que si bien tenemos algunas features con importancia mucho mayor, en general, se le da importancia a varias features, y no se observa el fenómeno de la *long-tail*, lo que a priori parece indicar que el trabajo de feature engineering realizado fue exitoso y que al modelo le sirvieron las featurea agregadas.

Algo que resulta interesante marcar en este gráfico, es que si bien la distribución es bastante equitativa, se puede observar que en el top 10 de features más importantes, el **idzona** aparece varias veces, con distintos métodos de encoding, lo que nos lleva a pensar que el barrio resulta ser el feature más importante para nuestro modelo.

7.2 XGBoost

Se grafica a continuación la importancia de las 25 features más importantes según **XGBoost**:

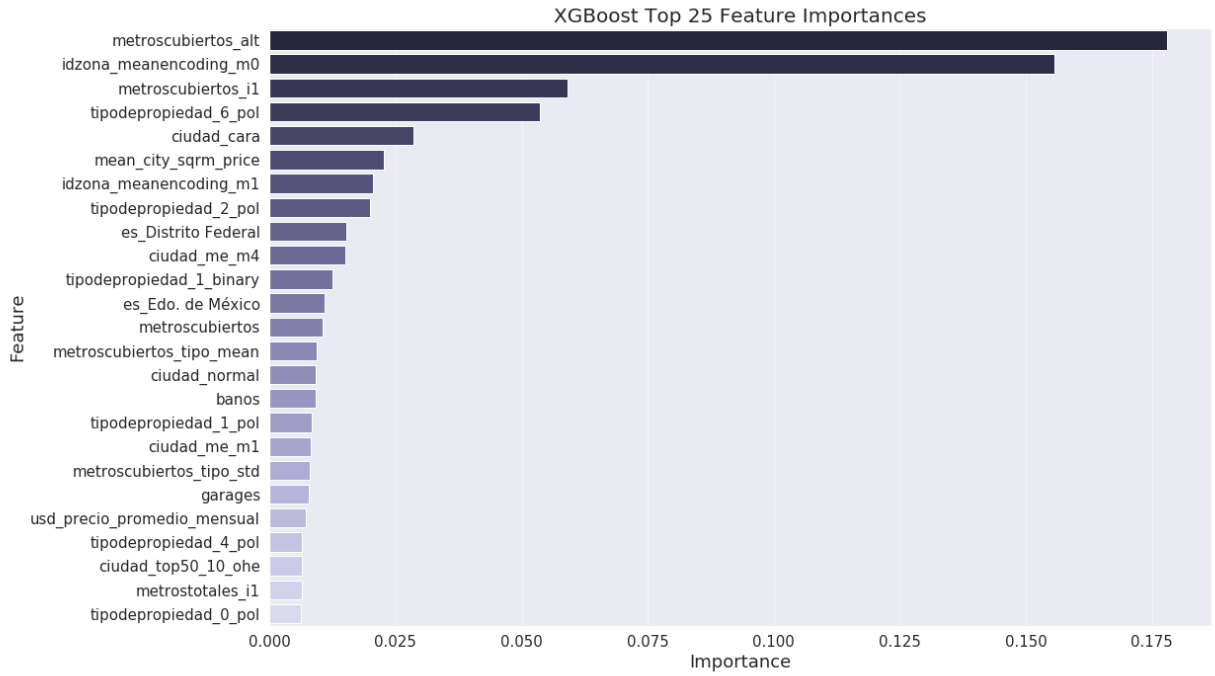

Figura 3: Feature Importance para XGBoost

Podemos apreciar en este gráfico que XGBoost le da mucha importancia a unos pocos features, generando una *long-tail* de features que aportan información, pero que tienen poca influencia sobre el resultado final.

A continuación, enumeramos algunas de las features más importantes para este modelo:

1. **metroscubiertos_alt**: tiene mucho sentido, como suponíamos desde un principio, los metros suelen ser el factor diferencial más importante a la hora de valuar una propiedad.

2. **idzona_meanencoding_mo**: el barrio resulta ser la segunda feature más importante, cosa que también tiene mucho sentido.

3. **metroscubiertos_i1**: al parecer, los metroscubiertos son tan importantes que toma información de los mismos de dos alternativas distintas que propusimos en feature engineering.

7.3 RandomForest

Se grafica a continuación la importancia de las 25 features más importantes según **RandomForest**:

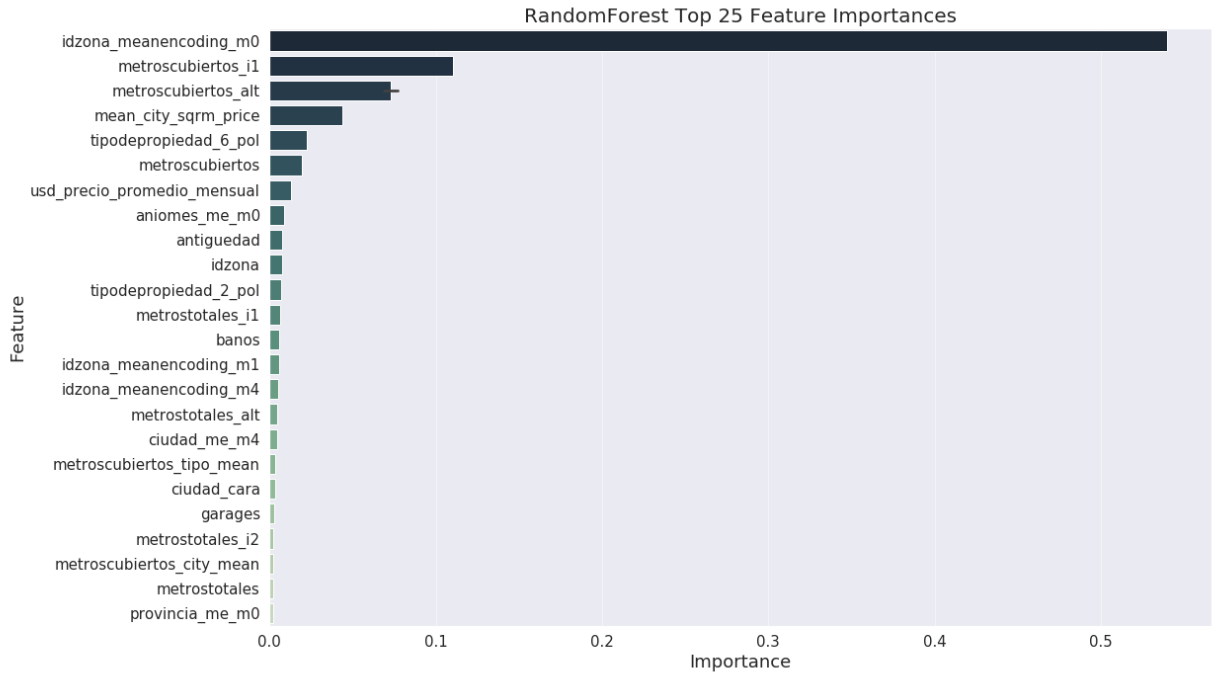

Figura 4: Feature Importance para RandomForest

En este caso, totalmente distinto a lo observado en el modelo **LightGBM**, podemos ver que se genera una *long-tail* muy marcada de features que no aportan prácticamente nada al modelo. De hecho es importante recordar que sólo se grafican las 25 features mas importantes, de las 150 con las que se entrenó al modelo, por lo que podemos ver que son sólo unas 10 features que realmente aportan información útil.

Sin embargo, como explicamos anteriormente, realizamos un proceso conocido como **Cumulative Importance** para RandomForest, con el objetivo de encontrar un menor número de features que generen resultados con menos ruido, y fue sólo en el caso de este modelo que el algoritmo nos funcionó. Graficamos a continuación, las 25 features más importantes pero una vez aplicado el algoritmo de selección:

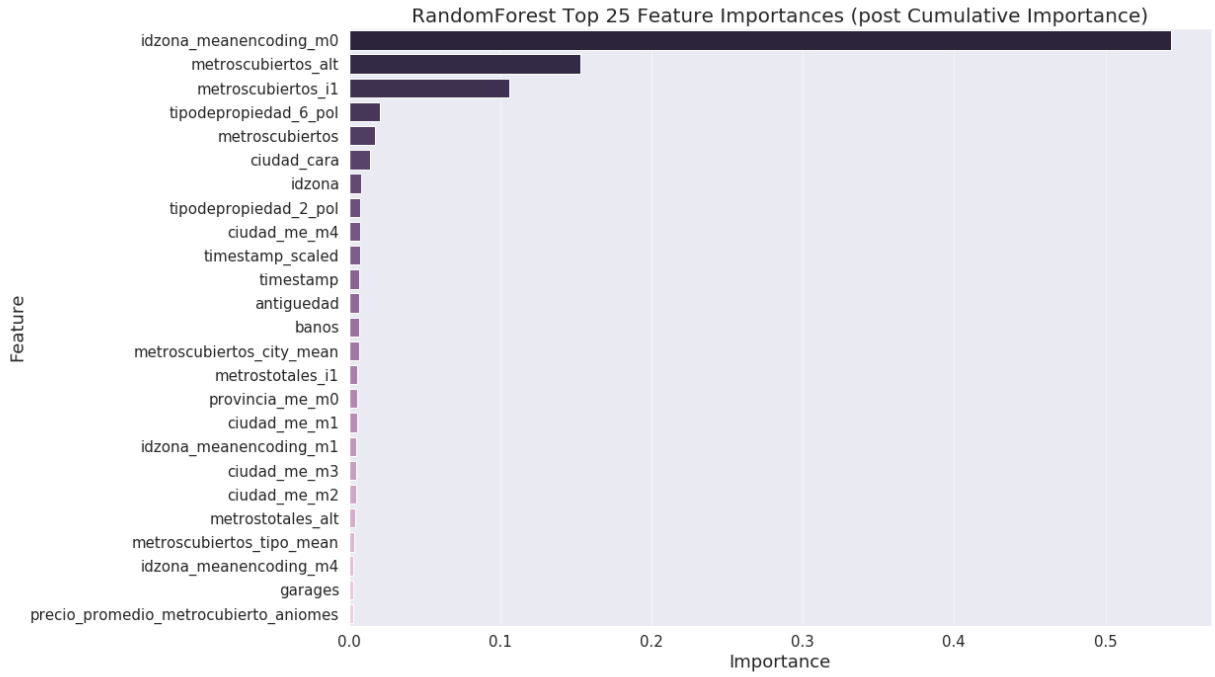

Figura 5: Feature Importance para RandomForest después de aplicar Cumulative Importance

Vemos que no se observan grandes diferencias en cuanto a la distribución de las importancias, aunque sí en cuanto a las features en sí.

8 MODELOS DE MACHINE LEARNING

Para encarar el problema de regresión planteado, se probaron una variedad de modelos de machine learning, con el fin de hallar el que diera los mejores resultados. Todos los resultados que se informan a continuación son utilizando una porción del 80 % de los datos de entrenamiento para entrenar el modelo y el resto de los 20 % como datos de validación. Es respecto de este conjunto de validación que se calculó el Mean Absolute Error informado.

8.1 Regresión lineal

8.1.1 *Breve introducción teórica*

La primera aproximación que suele darse a todos los problemas de regresión es ajustar los puntos del conjunto de entrenamiento con una recta, y utilizar esta recta para predecir las etiquetas de los puntos del conjunto de test.

8.1.2 *¿Por qué regresión lineal?*

La regresión lineal nos brinda información respecto del problema. Si las etiquetas que se desean predecir tienen una relación lineal con las features, entonces las predicciones generadas por el modelo pueden aproximarse con bastante certeza a los valores verdaderos.

Por otra parte, es un modelo fácil de lograr, y de entrenamiento muy rápido, por lo que es una buena primera aproximación a resolver el problema.

8.1.3 *Resultados obtenidos*

Mean Absolute Error = 736737.71
Como se puede apreciar, las predicciones no son demasiado buenas. Esto nos indica que no existe una tendencia lineal entre las features y el precio de las propiedades.

8.2 KNN

8.2.1 *Breve introducción teórica*

El modelo K-Nearest Neighboors consiste en pensar cada uno de los datos de entrenamiento como un punto. Cuando se desea predecir el tag para un cierto dato de test, se ubica este dato como un punto en el mismo espacio donde están los puntos de train, y se calcula cuales son los K puntos de train mas cercanos a este. Finalmente, si se trata de una variable categórica, la predicción es la categoría con mas ocurrencias entre los vecinos. Si se trata de una variable continua, la predicción es el promedio ponderado del label para los K vecinos, usando como pesos la inversa de la distancia de dichos vecinos al dato que se desea predecir.

8.2.2 *¿Por qué KNN?*

KNN Suele ser un buen enfoque a los problemas de clasificación, y quizás lo fuera también a los problemas de regresión, suponiendo que hubiera una cierta coherencia en la forma en que se clusterizan las propiedades según sus features y precio.

8.2.3 *Resultados obtenidos*

Mean Absolute Error = 809110.83
Los resultados obtenidos no son muy buenos. Quizás con un tuneo mas fino de los hiper-parámetros se pudiera haber obtenido un resultado mejor.

8.3 RandomForest

8.3.1 *Breve introducción teórica*

Random Forest consiste en aplicar una técnica de bagging a un modelo de árboles de decisión. Dado un conjunto de entrenamiento con sus respectivas labels, se generan sub-particiones de este conjunto tomadas con repetición, y se entrena con cada una de estas sub particiones un árbol de decisión. La característica distintiva de Random Forest es que, además de la sub-partición de datos, se utiliza en cada árbol un sub-set de n features tomados de forma aleatoria.

8.3.2 *¿Por qué RandomForest?*

Es un modelo muy eficiente, y es muy bueno a la hora de evitar overfitting. Además, su tuneo de hiper-parámetros es relativamente poco costoso.

8.3.3 *Resultados obtenidos*

Mean Absolute Error = 588415.48

8.4 XGBoost

8.4.1 *Breve introducción teórica*

Es un modelo de gradient boosting basado en árboles. Consiste en utilizar una sucesión de predicciones, en la cual cada modelo sucesivo intenta predecir el error del anterior. La predicción final resulta de la sumatoria de predicciones.

8.4.2 *¿Por qué XGBoost?*

Es el modelo estado del arte en estructuras de datos tabulares (como es el caso de este trabajo práctico). Algunos de los mejores resultados obtenidos fueron utilizando XGBoost.

8.4.3 *Resultados obtenidos*

Mean Absolute Error = 475474.81
Este resultado es muy cercano al mejor resultado obtenido con LightGBM.

8.5 LightGBM

8.5.1 *Breve introducción teórica*

Es otro modelo de boosting basado en árboles de decisión, pero con un enfoque prioritario en la eficiencia.

8.5.2 *¿Por qué LightGBM?*

Su eficiencia a la hora de entrenar lo convierten en un modelo muy práctico a la hora de probar introducir nuevas features, debido a que permite determinar de forma rápida si la feature agregada le aporta información nueva al modelo o no. Además, la diferencia de error obtenida respecto a XGBoost no es significativa.

8.5.3 *Resultados obtenidos*

Mean Absolute Error = 449620.60
Este fue el mejor submit obtenido por el grupo, y que obtuvo 472k de Mean Absolute Error en la competencia en Kaggle. Atribuímos la diferencia al fenómeno de **over-fitting**, que fue el principal problema de este modelo.

8.6 Redes Neuronales

8.6.1 *Breve introducción teórica*

Las redes neuronales son, en esencia, un conjunto de operaciones lineales que se realizan sobre el vector de datos de entrada, y que dan como salida un conjunto de probabilidades, donde cada una de estas probabilidades representa la probabilidad de que el dato que se desea predecir pertenezca a cada label.

8.6.2 *¿Por qué redes neuronales?*

A pesar de que no son muy eficientes para realizar regresiones, pueden ser útiles a la hora de generar features. Por ejemplo, para clasificar las propiedades en un cierto rango de precio (como se explicó en otra sección de este informe).

8.6.3 *Resultados obtenidos*

Mean Absolute Error = 621982.19
Este resultado, obtenido al utilizar un Multi Layer Perceptron Regressor, no es muy bueno en comparación al obtenido con los algoritmos de Boosting.

9 TUNEO DE HIPER-PARÁMETROS

El tuneo fino de hiper-parámetros se realizó sobre los modelos que fueron mas exitosos a priori. Estos modelos fueron LightGBM y XGBoost.

En un primer momento, se plantearon conjuntos de posibles hiper-parámetros para cada uno de los modelos, en intervalos originados a partir de la prueba de combinaciones a mano. Sobre estos hiper-parámetros, se intentó realizar GridSearch pero debido a la gran cantidad de tiempo que este proceso llevaba, se decidió utilizar en su lugar RandomSearch con una cierta cantidad de iteraciones.
En cada iteración del RandomSearch realizado, se realizó el proceso de K-Fold Cross-Validation para obtener una mayor precisión a la hora de elegir los hiper-parámetros óptimos.Se utilizo en este caso K=4.

Parametro	Opciones
max_depth	13, 14, 15
n_estimators	120, 130, 140
learning_rate	0.05, 0.1, 0.3
subsample	0.5, 0.8, 0.9
min_child_weight	15, 20

Cuadro 1: Tabla de hiper-parámetros de XGBoost.

Parametro	Opciones
num_leaves	55, 60, 65
max_depth	8,10,12
min_gain_to_split	0.1, 0.2
max_bin	50, 100, 150
min_data_in_leaf	3000, 5000, 7000
bagging_freq	4,5,6
bagging_fraction	0.65, 0.7, 0.75
feature_fraction	0.7

Cuadro 2: Tabla de hiper-parámetros de LightGBM.

La búsqueda aleatoria sobre los hiper-parámetros de LightGBM se realizo con 120 iteraciones, y llevo aproximadamente 17 horas.

La búsqueda aleatoria sobre los hiper-parámetros de XGBoost se realizo con 20 iteraciones y llevo aproximadamente 8 horas.

Es notable la diferencia de tiempo de entrenamientos entre XGBoost y LightGBM. Si se hubieran utilizado 120 iteraciones para XGBoost, este proceso hubiera llevado alrededor de 48 horas.

10 ENSAMBLES DE MODELOS

10.1 Blending entre XGBoost y LightGBM

Primero se realizo una aproximación muy simple, que consistió en generar predicciones utilizando primero XGBoost, agregar estas predicciones como una nueva feature al dataset, y luego entrenar un modelo LightGBM utilizando este dataset que incluye la predicción de XGBoost como feature. Se obtuvo un Mean Absolute Error de 488.000 contra un conjunto de validación de 20 % del set de entrenamiento, pero encontramos que la diferencia entre las predicciones contra el conjunto de validación y contra el conjunto de train era muy alta, dando muestras de overfitting.

Luego de este intento, se procedió a realizar lo que se conoce como *blending*, siguiendo el siguiente procedimiento:

1. Separamos el set de train en 90-10.
2. Entrenamos XGBoost y LightGBM con el 90 % del train set separado anteriormente.
3. Realizamos las predicciones sobre el 10 % restante.
4. Con estas predicciones, entrenamos un modelo blender, que puede ser un sencillo regresor lineal. En nuestro caso, decidimos utilizar nuevamente XGBoost.
5. Ahora se re-entrenan los modelos de XGBoost y LightGBM utilizando todo el set de train (más datos implica mejores resultados siempre) para luego predecir los precios del set de test.
6. Con estas predicciones, se alimenta al modelo blender, que nos dará las predicciones finales.

Con este método, obtuvimos un MAE aproximado de 500k en Kaggle.

10.2 Bagging con LightGBM

Se realizó un experimento muy simple también, utilizando la API proporcionada por *sklearn*. **Bagging** consiste en utilizar un mismo modelo varias veces para luego promediar sus predicciones. Obviamente, para que el modelo genere producciones distintas, debemos entrenarlo con distintos sets de train. La idea es que ningún modelo conoce la totalidad del set, sino que se generan subconjuntos del mismo tamaño que el set original, pero con reemplazo. Los resultados parecían ser buenos, logrando aproximadamente 430k de MAE, pero cuando realizamos el submit en Kaggle nos dió 500k, dando muestra de un claro overfitting. Probablemente si hubieramos dedicado más tiempo a hilar fino en este método y a probar con distintos hiper-parámetros hubieramos obtenido mejores resultados.

En una próxima sección del informe, se desarrollarán ideas respecto de los ensambles de modelos que nos hubiera gustado implementar, pero que por problemas de tiempo no fue posible.

11 RESULTADOS

Los mejores resultados obtenidos en Kaggle fueron logrados utilizando LightGBM, las features seleccionadas, y el siguiente conjunto de hiper-parámetros:

- max_depth = 14
- num_leaves = 120
- learning_rate = 0.02

Este submit obtuvo un Mean Absolute Error de aproximadamente **472000**, pero a la hora de realizar una comprobación contra el subconjunto de entrenamiento, encontramos que había una gran diferencia entre ambos errores, lo cual es un gran indicador de que el modelo estaba sumamente overfitteado. Fue luego de ver este resultado que se decidió hacer una búsqueda de hiper-parámetros que lograran un menor overfitting sobre los datos, ya que como sabemos el overfitting es un fenómeno que produce mucha varianza en nuestras predicciones, algo que es muy desfavorable en un modelo.

Para lograr controlar el overfitting, utilizamos **GridSearch** y **RandomSearch**, llegando al siguiente conjunto de hiper-parámetros:

- num_leaves = 55
- min_gain_to_split = 0.2
- min_data_in_leaf = 3000
- max_depth = 12
- max_bin = 150
- feature_fraction = 0.7
- bagging_freq = 5
- bagging_fraction = 0.75
- n_estimators = 5000
- learning_rate = 0.05

Tras haber trabajado para lograr reducir el mismo, pudimos obtener, mediante un blending entre LightGBM y XGBoost, un resultado que rondó los 500k en Kaggle. Este resultado es peor que el obtenido utilizando LightGBM con los parámetros que mencionamos en el inicio de esta sección, pero logramos reducir el overfitting del modelo, por lo que se puede afirmar que este otro resultado tiene una varianza mucho menor y por lo tanto es más confiable. Probablemente de haber continuado con las pruebas con ensambles y con el tuneo de parámetros para evitar el overfitting habríamos logrado mejores resultados, como hablaremos en la próxima sección.

12 IDEAS QUE NO PROSPERARON

A lo largo del desarrollo de este trabajo práctico, se nos fueron ocurriendo distintas ideas para mejorar la precisión de nuestros modelos. Si bien muchas de ellas produjeron mejorías notorias, y otras mejorías de menor índole, también hubieron **ideas que si bien a priori parecían ingeniosas, empeoraron la precisión del modelo**. Se detallan a continuación algunas de estas.

12.1 Dividir el problema

Esta idea nació en base al **análisis de error** que decidimos realizar una vez que tuvimos resultados razonablemente buenos. Para esto, aplicamos nuestro modelo para predecir los precios de las propiedades de train, y luego calculamos el error absoluto en cada publicación. En base a este calculo, intentamos encontrar patrones en los tipos de propiedades que estaban generando un error mayor, y encontramos los siguientes problemas muy marcados:

- **Distrito Federal**: comparamos el error absoluto promedio (MAE) entre las propiedades que eran de Distrito Federal y las que no, y la diferencia era de **350.000** aproximadamente, lo que nos pareció un número muy elevado teniendo en cuenta que nuestro mejor MAE en ese momento rondaba los 500k.
- **Diciembre de 2016**: realizamos una comparación similar para las propiedades de Diciembre del 2016 (del TP1 sabemos que hay muchas más publicaciones este mes que los demás), encontrando nuevamente un error mucho mayor: mas de 200k de diferencia.
- **Propiedades caras**: también notamos que las propiedades que tenían precios mayores al precio promedio + el desvío estandar, también estaban teniendo un error mucho mayor a las que no.
- **Datos sin sentido**: finalmente analizamos los datos que estaban teniendo un error mucho mayor que el mejor MAE conseguido hasta el momento, y encontramos que teníamos muchas propiedades que estaban teniendo errores mayores a los 5.000.000, cuando el mejor MAE era de 500k. Estas propiedades eran justamente datos sin sentido, o que tenían practicamente todos los datos faltantes.

12.1.1 *Primera idea: clasificar outliers y datos sin sentido*

Una primer idea fue separar el problema en dos: antes de comenzar a entrenar modelos de regresión, aplicar un clasificador que se encargue de identificar outliers y datos sin sentido, siguiendo una serie de criterios que establecimos. Una vez clasificados, separamos el dataframe en dos, por un lado los datos confiables y por otro los datos peligrosos. Nuestros modelos de regresión sobre los datos confiables estaban dando predicciones de 350k de MAE, lo que nos pareció una mejoría muy notoria, pero sobre los datos peligrosos lograbamos MAEs muy malos, del órden de los 900k de error absoluto promedio. Cuando juntamos ambos modelos para la predicción final, no pudimos mejorar el mejor MAE que teníamos hasta el momento, por lo que concluímos que si bien la idea de dividir el problema tenía mucho sentido, probablemente clasificar outliers era una tarea mucho más dificil y muy engañosa, y la precisión que creíamos tener para lograrlo no era tal. Desechamos este modelo.

12.1.2 *Segunda idea: clasificar precios altos*

Siguiendo en la misma linea, la siguiente idea que tuvimos fue intentar clasificar algo más razonable que outliers, que pueden ser difíciles de identificar. Ahora el

plan era el siguiente: primero entrenabamos un modelo de clasificación para predecir si una determinada propiedad tendría un precio mayor al precio promedio + desvío (sabíamos que estas propiedades confundían al modelo), con el objetivo de poder advertirle al modelo que se trataría de un dato con un precio elevado. Si bien esto parecía tener buenos resultados, nuevamente, el problema fue que el modelo de clasificación introducía un error bastante grande, generando que se pierda la ganancia obtenida cuando el mismo predecía bien. Desechamos este otro modelo.

12.1.3 *Tercer idea: separar datos Diciembre 2016*

Cambiando un poco de foco, ahora intentamos tratar las anomalías generadas por los datos de **Diciembre del 2016**, que sabíamos que nos estaban generando un error mayor al promedio. Para esto, propusimos algunas soluciones, aunque ninguna logró darnos buenos resultados:

- **Separar en dos datasets**: al igual que con los outliers, intentamos separar el problema en dos, pero el error al juntar ambos regresores seguía siendo exactamente el mismo, y hasta empeoraba dependiendo de los hiper-parámetros.
- **Inducir valores nulos**: aunque pueda parecer raro, una idea que se nos cruzó por la cabeza fue la de rellenar con NaNs los campos fecha de las publicaciones de Diciembre del 2016. Esta idea se basaba en que teníamos la creencia de que muchas propiedades no eran de ese momento temporal, pero que se habían mandado ahí por no tener la información precisa o por querer truncar el dataset hasta 2016. Además, sabemos que el manejo de nulos que realizan los modelos de Boosting como son XGBoost y LightGBM es, en general, muy bueno y mejor que cualquier imputación. Sin embargo, los resultados no fueron positivos.

13 IDEAS QUE NO FUERON IMPLEMENTADAS

Además de ideas que fueron implementadas pero que no funcionaron, también hubieron una serie de ideas que simplemente no pudieron ser puestas a prueba por falta de tiempo o por poca feasibilidad.

13.1 Crawler

Una idea surgida de una sesión de brainstorming fue implementar un crawler para poder obtener mas información de las propiedades de los conjuntos, pero esta idea se dejó de lado debido a que hubiera sido demasiado costoso buscar información relacionada a cada una de las propiedades utilizando su dirección o su latitud y longitud, y luego realizar un procesamiento de todos esos datos para poder obtener features.

13.2 API de Google Maps

Google Maps provee una API que permite obtener cierta informacion respecto de lugares teniendo únicamente su dirección. Desafortunadamente, esta API es paga, y parece no ser trivial utilizarla para obtener información respecto al entorno que rodea a las propiedades (como podrían ser comercios cercanos, servicios, etc).

13.3 Ensambles

No fue posible hacer una gran cantidad de ensayos de ensambles, y creemos que esto nos hubiera llevado a muy buenos resultados. Una posible mejoría sería realizar lo que se conoce como **blending**, un modelo que "mezcla" los resultados obtenidos por distintos algoritmos, dándoles distinta importancia a cada uno de ellos. Si bien experimentamos con una primera aproximación utilizando XGBoost y LightGBM, los resultados no fueron buenos. Nos hubiera gustado poder implementarlo utilizando más modelos y probando distintos blenders para finalizar las predicciones. Otra posibilidad hubiera sido realizar Averaging entre estos mismos tres modelos (dado que son los que mejores resultados ofrecieron).

14 CONCLUSIONES

Consideramos que este trabajo fue muy **formativo**, ya que nos permitió transitar de principio a fin un camino muy interesante en el que, con el objetivo final de predecir los precios de las propiedades, no sólo tuvimos que probar modelos sino intentar **entender la naturaleza de los datos** y buscar soluciones ingeniosas a problemas que se fueron presentando.

Aprendimos que si bien por lo general para cada tipo de problema existe un modelo que predomine como el "mejor", nunca está de más probar distintas alternativas, porque muchas veces pueden generar resultados inesperados.

También nos resultó muy interesante todo el proceso de **Feature Engineering**, en el que pudimos proponer distintas formas de representación de la información que poseíamos con el objetivo de facilitarle al modelo datos que no podía deducir inicialmente. Se utilizaron distintas técnicas que intentaron ser creativas para poder lidiar con algunos problemas puntuales, como el caso de cómo equidistanciar los meses (la distancia entre Diciembre y Enero debería ser la misma que entre cualquier otro par de meses, a lo que se propuso aplicar una *transformación circular* como es el coseno. Algo que nos llamó mucho la atención fue que nuestros mejores saltos de precisión se debieron muchas veces a factores de los que no esperabamos tal mejoría, mostrando que muchas veces existen atributos mucho mas valiosos de lo que pensamos a primera vista. A continuación, listamos algunos de estos casos con los que nos topamos a lo largo del desarrollo:

- **Feature Selection**: si bien sabíamos que seleccionar los mejores features aumentaría la precisión del modelo, nos sorprendimos al ver que pudimos pasar de más de 300 features a unas 140, disminuyendo en más de 20k el error que se estaba cometiendo en ese momento. Esto fue una clara muestra de que una parte importante de cualquier modelo de Machine Learning radica en **eliminar el ruido** que puede confundirlo.

- **Inclusión del idzona**: inicialmente, por una confunsión en el entendimiento de qué significaba esta columna, decidimos no incluirla en el entrenamiento, resultando en un MAE aproximado de 570k, en el mejor de los esfuerzos. Cuando decidimos incluirla, el salto fue de más de 50k, llegando a los 520k de un submit a otro.

- **Tuneo de hiper-parámetros**: nuevamente, otro factor que no creímos tan importante y que resultó en cambios abismales. Si bien sabíamos que los hiperparámetros son muy importantes en cualquier modelo, no creímos que podrían transformar un modelo que a priori parecía ineficiente para el problema, en un modelo que termine siento utilizado en la predicción final, como sucedió con **LightGBM**. Inicialmente, con los parámetros por defecto, conseguimos resultados muy malos, siendo éstos muy inferiores a los obtenidos con **XGBoost**, por lo que se decidió no incorporar este modelo. Sin embargo, cuando empezamos a probar parámetros de forma manual y luego de forma automatizada mediante GridSearch con Cross-Validation, obtuvimos una mejoría tal que decidimos incorporarlo.

- **Manejo de nulos en métodos de Boosting**: algo que nos llamo mucho la atención fue el eficiente manejo de valores nulos que realizan los modelos tales como **XGBoost** y **LightGBM**. Inicialmente, creímos que era vital imputar los nulos de alguna manera, probando distintas alternativas desde el promedio hasta intentar rellenar estos valores utilizando otro regresor. Sin embargo, nos dimos cuenta que estos modelos funcionaban mucho mejor sin esta imputación. Otro factor que no tuvimos en cuenta inicialmente y mejoró los resultados, en contra de lo esperado.

Obviamente no todo sucedió como se esperaba, tuvimos muchas ideas que a priori parecían tener mucho sentido y mucha base teórica, pero que al fin y al cabo resultaban en **predicciones menos precisas**, confundiendo al modelo o agregando ruido. Llegamos a la conclusión de que los modelos aprenden de los datos mucho más de lo que nosotros creemos, y que son capaces de encontrar relaciones que visualmente no se observan a primera vista. Es por esto que muchas veces que creíamos estar agregando información, en realidad esa información ya era conocida por el modelo, y nosotros simplemente agregabamos ruido.

Para finalizar, nos gustaría aclarar que si bien se considera que se realizó un trabajo muy extenso y de gran profundidad, el mismo podría ser continuado desde donde se lo dejó, probando distintos métodos de selección de features, tuneando aun más los hiper-parámetros, y probando distintos ensambles de modelos que por cuestiones de tiempo no pudimos implementar. Aun así, consideramos que los resultados obtenidos son muy buenos, y nos llevamos mucha experiencia en el área que consideramos que nos será muy útil para la próxima vez que encaremos un problema de este tipo.