

Trabajo Práctico N° 2: Competencia de Machine Learning

Precios de Propiedades

Organización de Datos [75.06 / 95.58]
Segundo Cuatrimestre 2019

Grupo N° 43
DataAttack

Nombre y Apellido	Padrón
Soledad Escobar	97877
Luciano Ortiz	100323
Fernando Moreno	96683

Fecha de Entrega: 2 de Diciembre de 2019

Índice

Introducción	3
Objetivo del Trabajo Práctico	3
Sobre la Empresa	3
Sobre los Datos	3
Limpieza de los datos	5
Tratamiento de valores nulos	5
1º Enfoque:	5
Título, descripción y dirección	5
Tipo de propiedad	5
Latitud y longitud	5
Provincias y ciudades	5
Metros cubiertos	5
Habitaciones	6
Baños	6
Id zona	6
Garages y antigüedad	6
Metros totales	6
2º Enfoque:	6
Habitaciones:	7
Baños:	7
Garages:	7
Provincia:	7
Ciudad:	7
Direccion e idzona:	7
Tipodepropiedad:	7
Metroscubiertos:	7
Metrostotales:	8
Lat y Ing:	8
Antiguedad:	8
Titulo y descripcion:	8
Fecha, gimnasio, usosmultiples, piscina, escuelascercanas, centroscomercialescercanos, precio:	8
Feature Engineering	8
Features sobre variables categóricas	8
One Hot Encoder	9
Target Encoder	10

Competencia de Machine Learning ZonaProp - Organización de Datos

Metros Cubiertos y Metros Totales	10
Cantidad de publicaciones	11
Baños	11
Garages	12
Habitaciones	13
Antigüedad	13
Características: Gimnasio, Piscina, Usos Múltiples, Escuelas y Centros Comerciales	
Cercanos	14
Direcciones	15
CatBoost Encoder	15
Features Sobre el tiempo	15
“Puntaje por fecha”	16
“Publicaciones por día / mes / año”	17
Features creados a partir del uso de SVD	17
Nuevos features a partir de la reducción del set de features	17
SVD sobre One Hot Encoding	19
Ciudades	19
Provincias y tipos de propiedad	20
Features sobre el Precio (precio_min y precio_max)	21
Rango de precio esperado según la ciudad	21
Rango de precios esperado según provincia y fecha de la publicación	22
Features Sobre Palabras en Títulos y Descripciones	22
Palabras importantes:	22
En título:	22
En descripción:	23
Modelos de Machine Learning	25
Modelos Baseline	25
Linear Regression	25
KNN Regressor	25
Decision Tree Regressor	27
Random Forest	27
Gradient Boosting	28
XGBoost	29
AdaBoost Regressor	30
Voting Regressor	31
Lasso sobre XGBoost y Random Forest	32
Conclusiones	33
Importancia de Features	33
Feature Importance Random Forest	33
Feature Importance XGBoost	34
Conclusiones Finales	34

Links a GitHub	36
Link al Trabajo Práctico Número 2:	36
Link al Trabajo Práctico Número 1:	36

Desarrollo del Trabajo Práctico

Introducción

Objetivo del Trabajo Práctico

El presente trabajo práctico corresponde a una competencia de Machine Learning, realizada a través de la plataforma Kaggle, en donde se intentará predecir, para cada propiedad, el precio de la misma con la mayor precisión posible.

Para conseguir este objetivo aplicaremos los conocimientos adquiridos a lo largo de la cursada aplicando y probando el resultado de las diversas técnicas de Feature Engineering y combinando el uso de diferentes modelos de Machine Learning.

Sobre la Empresa

ZonaProp es el mayor portal de compra y venta de inmuebles.

El portal cuenta con la mayor variedad de casas, departamentos, oficinas comerciales y más. Cuenta con una página web y una aplicación. ZonaProp permite buscar inmuebles filtrando las publicaciones de acuerdo a los requisitos de cada usuario, pudiendo seleccionar si desea comprar, alquilar, vender y además indicar el tipo de inmueble. También puede filtrarse de acuerdo a la ubicación, como ciudad, barrio, etc., o de acuerdo a la cantidad de ambientes y lo más importante, de acuerdo al precio, permitiendo así que cada usuario sólo vea las publicaciones acerca de inmuebles a los que puede acceder de acuerdo al rango de valores que puede permitirse.

Sobre los Datos

El dataset de entrenamiento consta de propiedades en venta en México entre los años 2012 y 2016, valuadas en pesos mexicanos. El archivo train.csv tiene 240K filas y 22 columnas.

- id: Un id numérico para identificar la propiedad
- titulo: El título de la propiedad publicada
- descripcion: La descripción de la propiedad publicada
- direccion: La dirección de la propiedad

Competencia de Machine Learning ZonaProp - Organización de Datos

- ciudad: La ciudad de la propiedad
- provincia: La provincia donde está localizada la propiedad
- lat: Latitud
- Ing: Longitud
- tipodepropiedad: El tipo de propiedad (Casa, departamento, etc)
- metrostotales: Metros totales de la propiedad
- metroscubiertos: Metros cubiertos de la propiedad
- antiguedad: Antigüedad de la propiedad
- habitaciones: Cantidad de habitaciones
- garages: Cantidad de garajes
- banos: Cantidad de baños
- fecha: Fecha de publicación
- gimnasio: Si el edificio o la propiedad tiene un gimnasio
- usosmultiples: Si el edificio o la propiedad tiene un SUM
- piscina: Si el edificio o la propiedad tiene un piscina
- escuelascercanas: Si la propiedad tiene escuelas cerca
- centroscomercialescerca: Si la propiedad tiene centros comerciales cerca
- precio: Valor de publicación de la propiedad en pesos mexicanos

Para realizar las predicciones contamos con un set de test, el cual consta de las mismas columnas que el set de entrenamiento pero no nos proporciona el precio de las propiedades, pues es lo que debemos predecir.

Limpieza de los datos

Antes de comenzar a crear features y probar modelos lo primero que se hizo fue examinar los datos para ver como podríamos levantarlos en memoria de manera más eficiente y para detectar la presencia de valores nulos en cada uno de los campos de los set de datos y reemplazarlos por los valores más convenientes en cada caso.

Tratamiento de valores nulos

1º Enfoque:

A continuación se explicará una de las estrategias que se tomaron para el llenado de valores nulos para cada columna y también la eliminación de valores anómalos detectados.

Título, descripción y dirección

Para estos tres campos no hubo ningún llenado especial, simplemente se utilizó un espacio en blanco o guión en el caso de dirección.

Tipo de propiedad

Aquí primero se contabilizaron los valores nulos y se encontraron que fueron pequeños, por lo tanto se completaron los faltantes utilizando los títulos y las descripciones, ya que en todos estos casos las mencionadas columnas permitían ver de qué tipo se trataba. Además se hizo un reacomodamiento de algunos tipos que tenían muy pocos valores, como "Lote" que se convirtió en "Terreno" y otros como "Hospedaje" y "Garage" se pasaron a "Otros", ya que estos dos solo aparecían en el train y no eran de utilidad de esa forma.

Latitud y longitud

Para las coordenadas, primero se eliminaron valores anómalos reemplazandolos por 0, en caso de que las latitudes no estuvieran entre 13 y 34 grados y las longitudes entre -84 y -119. Estos valores se obtuvieron como referencia mirando un mapa con coordenadas. Este tratamiento se hizo para luego poder completar algunos faltantes de provincias y ciudades que sí contaban con coordenadas.

Provincias y ciudades

Como se mencionó anteriormente, se llenaron algunos faltantes buscando la propiedad más cercana según latitud y longitud, y se completó con ese valor. Para el caso de las ciudades solo se buscaron valores donde la provincia fuera la misma. Como no todos los nulos de provincias y ciudades tenían coordenadas, al resto se los completó con 'N/A'

Metros cubiertos

En este campo se completaron los valores usando los promedios por tipo de propiedad y cantidad de habitaciones. En el caso de que una propiedad no tuviera registrado el número de habitaciones,

Competencia de Machine Learning ZonaProp - Organización de Datos

solo se usaron los promedios por tipo de propiedad. Para los casos donde el promedio era un valor nulo, se completó con el promedio general de metros cubiertos.

Habitaciones

En este caso, se calculó el promedio de habitaciones por metros cubiertos para cada tipo de propiedad y luego, para llenar el valor faltante, se dividió los metros cubiertos del faltante por el respectivo promedio y se redondeó el valor a un entero, siendo 1 el valor mínimo.

Baños

Aquí se calculó el promedio de baños por habitaciones para cada tipo de propiedad y se dividió la cantidad de habitaciones de la propiedad a llenar por el correspondiente promedio para llenar el nulo. En caso que el promedio diera menor a uno, se redondeará a uno.

Id zona

Los faltantes de este feature simplemente se rellenaron con cero.

Garages y antigüedad

En ambos casos se usaron los promedios por tipo de propiedad. Adicionalmente, para la antigüedad se llenó con 0 en los casos donde la palabra “estrenar” aparecía en el título o en la descripción.

Metros totales

Primero, se realizó una corrección sobre estos valores porque encontramos que había muchos casos donde los metros totales eran menores que los metros cubiertos, lo cual no tiene sentido, y en estos se reemplazó los metros totales con los metros cubiertos. Después, para llenar los faltantes, se usaron los promedios por tipo de propiedad y cantidad de habitaciones o solo por tipo de propiedad si no existía el promedio para cierta cantidad de habitaciones, pero siempre chequeando que el valor final fuera mayor o igual a los metros cubiertos, porque sino se completaba con este último.

2º Enfoque:

Se probaron distintas cosas al limpiar los datos y usarlos en los modelos que mejor score nos daban, pero algunas que a simple vista parecían buena idea terminaban empeorando las predicciones, algunos ejemplos son:

- Llenar nulos de lat y lng con el promedio de cada uno según su provincia o su ciudad.
- Llenar nulos de metros cubiertos y metrostotales con el promedio de cada uno según su tipodepropiedad y cant. de habitaciones, o solo según su cant. de habitaciones

Al final lo que mejor resultado terminó dando fue:

Habitaciones:

Había 22471 nulls en esta columna. El promedio total de cantidad de habitaciones del set de train es 2.90, redondeando para arriba nos queda que la mayoría tienen 3 habitaciones. Además si calculamos el precio promedio de las publicaciones que tienen null en habitaciones nos da cercano al precio promedio de las publicaciones con 3 habitaciones. Con estas observaciones decidimos llenar los nulos de esta columna con el valor 3 tanto en train como en test.

Baños:

26221 nulls en el set de train. Hicimos el mismo procedimiento que para habitaciones, y terminamos reemplazando estos nulls y los de train por el valor 2.

Garages:

Contaba con 37765 nulls en el set de train. El precio promedio de las publicaciones que tenían null en train para esta columna estaba entre el precio promedio de las que tenían 1 garage y las que tenían 2 garajes, pero era un poco más cercano al de las que tenían 2 garajes. Así que decidimos reemplazar todos los nulls de train y test para esta columna con el valor 2.

Provincia:

Había 155 nulls en provincia en el set de train. Rellenamos los nulls con el valor de provincia que más apariciones tiene en train. Los dos valores con más apariciones eran Edo. de México con 41607 (17% del total de train) y Distrito Federal con 58790 (24% del total de train). Decidimos llenar entonces los nulls de train y de test de esta columna por el valor Distrito Federal.

Ciudad:

Había 372 nulls en train. Decidimos hacer el mismo procedimiento anterior para ciudad, en este caso no era tan trivial elegir la ciudad ya que, la que más aparece, aparece solamente un 5% de veces en el total de train seguido de otras que aparecen un 4% de veces. Hay muchos valores de ciudades y no hay una que se destaque por aparecer muchísimo más que las otras. Sin embargo elegimos la que aparecía más veces, por mas que sea por poco, y esta era Querétaro.

Direccion e idzona:

A estos features no les dimos importancia a su llenado de nulls ya que luego no los usamos.

Tipodepropiedad:

Solamente 46 nulls en train. Rellenamos los nulls de esta columna en train y test con el tipodepropiedad que más aparece en train, que es por mucho 'Casa' (casi el 60% de las publicaciones en train son originalmente Casas)

Metroscubiertos:

Este feature es uno de los más importantes para el modelo a la hora de predecir, y sin embargo train originalmente tiene 17400 nulls. Decidimos reemplazarlos, tanto en train como test, por el valor promedio de metroscubiertos para todo el set de train. Este valor resultó ser 174.

Metrostotales:

Con el mismo procedimiento anterior, contábamos con 51467 nulls en train que en este caso fueron reemplazados por el valor 177, así como también los nulls del set de test.

Lat y Ing:

Estas eran las columnas que más nulls tenían, y sin embargo para el modelo estos features tenían bastante importancia. Se probó el enfoque de llenarlos de acuerdo a la provincia en que se encontraban, pero lo que mejor resultado dio de lo que probamos fue llenarlos con el promedio de cada una para el set de train. Entonces los nulls de lat (tanto de train como de test) fueron llenados con 20.69 y los de Ing con -99.50. Eran 123488 nulls en total en train, tanto para lat como para Ing.

Antiguedad:

Tenía 43555 nulls en train, los rellenamos a esos y a los de test con la antigüedad promedio de train, cuyo valor resultó ser 8.

Titulo y descripción:

Simplemente llenamos todos estos nulls con un guión, para no tener problemas luego a la hora de procesarlos.

Fecha, gimnasio, usosmultiples, piscina, escuelascercanas, centroscomercialescercanos, precio:

Todas estas columnas no tienen ningún null, así que simplemente las convertimos al tipo de dato que sea más conveniente para ahorrar memoria y poder trabajarlos de manera más eficiente.

Este segundo enfoque fue el que terminamos usando para la predicción final.

Feature Engineering

Para comenzar en desarrollo de la parte de Feature Engineering lo primero que realizamos fue tomar los valores numéricos que se encuentran proporcionados por el set de entrenamiento y utilizarlos como una base a la cual fuimos agregando los diferentes features que probamos.

Cabe mencionar que en esta sección se listan y explican todos los features desarrollados a lo largo del trabajo, esto incluye features que han funcionado y features que no han funcionado y que por consecuencia han sido descartados del set de features definitivo.

Features sobre variables categóricas

Competencia de Machine Learning ZonaProp - Organización de Datos

Contamos con varios campos que están compuestos por variables categóricas, como las provincias, ciudades, tipos de propiedad e incluso podría tomarse la idzona como una variable categórica, aunque viene como un tipo numérico, para poder utilizar la información que se encuentra en estos campos realizamos diferentes pruebas con “encoders” de diferentes tipos.

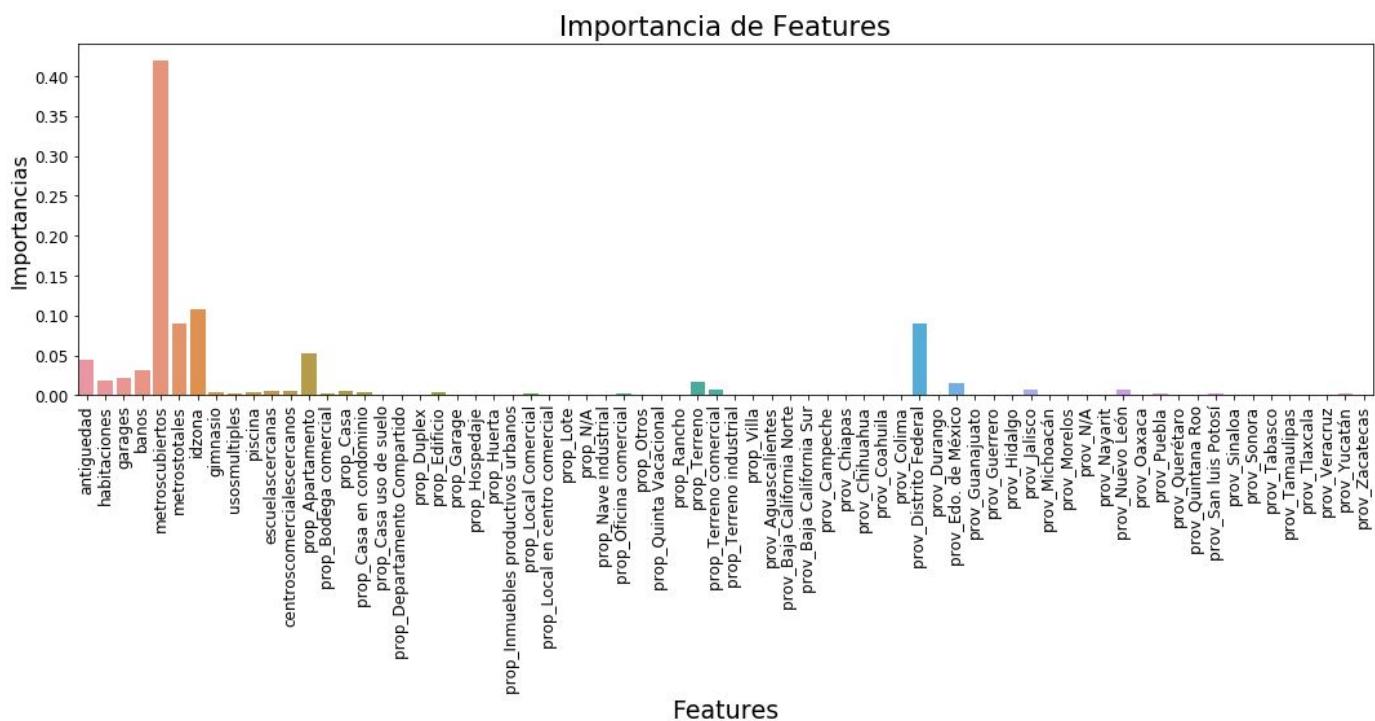
One Hot Encoder

Lo más intuitivo que pudimos hacer fue aplicar One Hot Encoding sobre las variables categóricas como tipo de propiedad, ciudad y provincia.

Aplicando este método generamos una columna por cada valor posible en cada uno de los campos mencionados, indicando con 1 y 0 si la propiedad pertenece o no a dicho atributo.

Es evidente que esto aumentó notablemente la cantidad de columnas que manejamos en el set de features, pues los tipos de propiedad cuentan con 25 posibles valores diferentes, las provincias con 33 diferentes valores posibles y las ciudades son más de 800 diferentes, por lo que llegamos a contar con más de 900 columnas dentro de nuestro set de features.

Este método nos dio resultados muy buenos, haciendo que nuestro score mejore bastante, pero veamos, para el caso de provincias y tipos de propiedad, cómo se comportaba la importancia de estos features en el modelo:



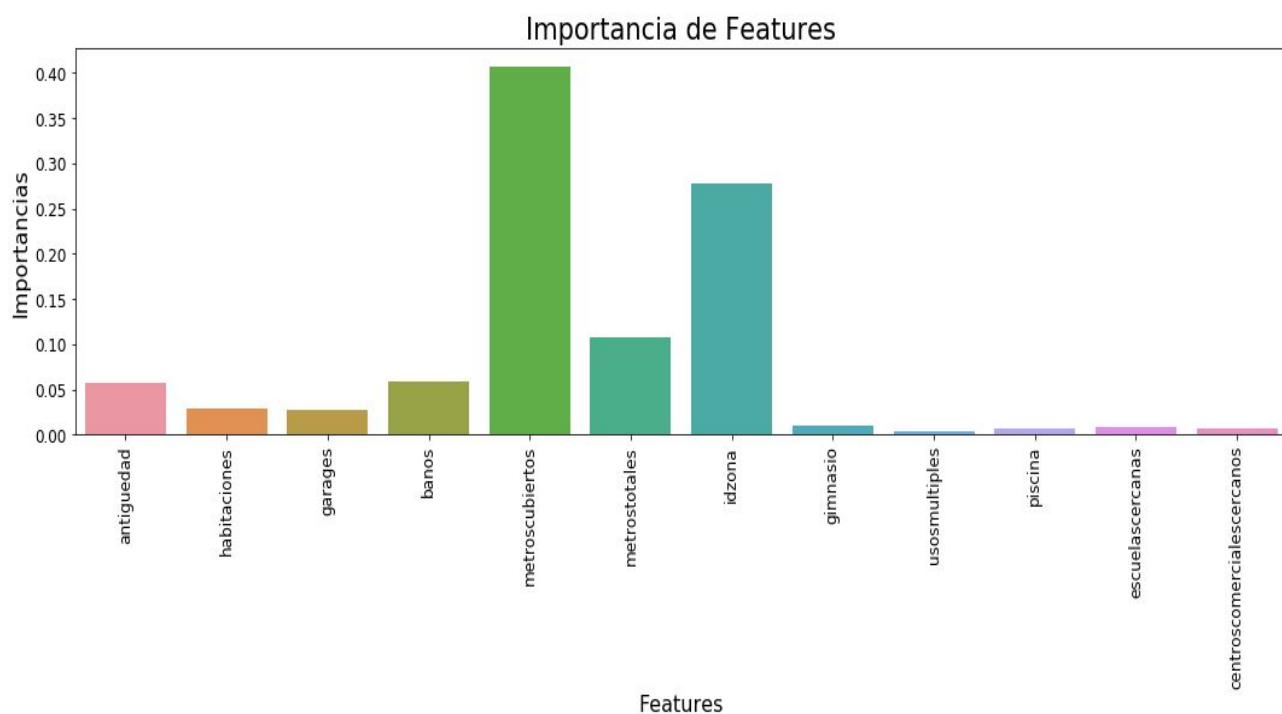
vemos que la mayoría tienen una importancia muy baja, pero en conjunto funcionan bien aunque el hecho de generar tantos features hizo que la prueba de cualquier modelo se torne bastante lenta, por lo que decidimos probar otros tipos de encoders donde se pudiera aprovechar mejor la información sin necesidad de crear un número tan elevado de features. Por supuesto que esto lo hicimos evaluando siempre como se veía afectado el score final luego de estos cambios.

Finalmente luego de probar con diferentes encoders, los cuales se detallaran en la siguiente sección, se logró mejorar los resultados sin necesidad de utilizar tantas columnas por lo que se decidió descartar estos features del modelo final.

Target Encoder

Una alternativa que encontramos para no utilizar una cantidad alevosa de columnas, como nos sucedió con el caso de One Hot Encoder, fue aplicar Target Encoder sobre los campos tipo de propiedad, ciudad, provincia e idzona, generando así una transformación de éstas columnas a valores numéricos que tengan relación con nuestros datos y nuestra variable a predecir, dicha transformación se realiza calculando un promedio del target de acuerdo a cada tipo de valor categórico existente en cada uno de los campos.

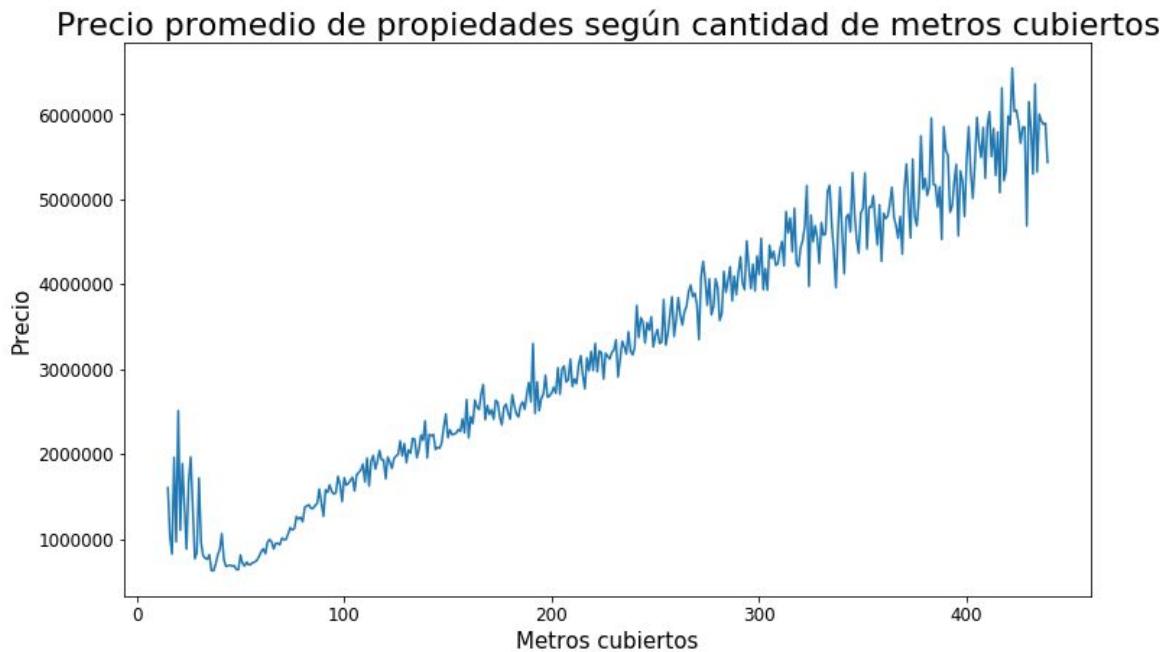
Antes de comenzar a describir cada uno de los targets utilizados veamos cuales son los features numéricos originales con los que contamos y la importancia de cada uno de ellos:



A simple vista observamos como los features más importantes son metros cubiertos, idzona y metros totales, a continuación detallamos como trabajamos y tratamos de aprovechar al máximo la información que nos proveen estos features.

Metros Cubiertos y Metros Totales

Durante el desarrollo del trabajo práctico número 1 descubrimos que la cantidad de metros que ocupaban las propiedades estaba fuertemente relacionado al precio por lo que utilizamos ese campo como target para este encoder, a continuación pueden observar dicha relación en un gráfico.



Como veíamos al comienzo de esta sección, en el gráfico de importancias, este feature era el más importante y esta es la razón, su comportamiento es muy parecido al del precio de las propiedades, lo cual tiene sentido pues es bastante lógico que, en la mayoría de los casos, cuanta más superficie ocupa la propiedad mayor sea su precio, por lo que decidimos aprovechar al máximo esta relación creando targets de acuerdo a los metros cubiertos en cada tipo de propiedad, ciudad, provincia e idzona.

Cantidad de publicaciones

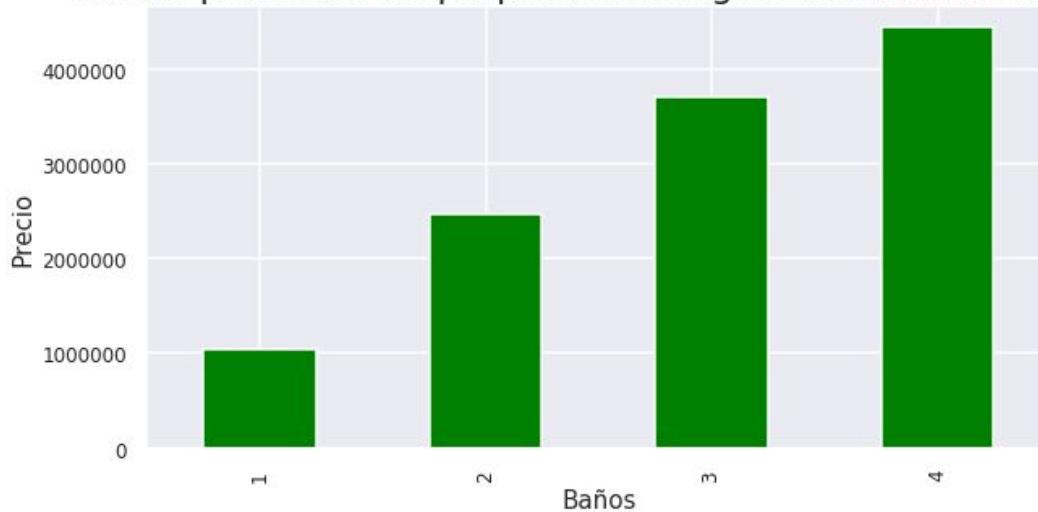
Otra alternativa para el target fue utilizar la cantidad de publicaciones realizadas, por año, por mes o por año, ya que también durante el trabajo práctico número 1, observamos que la cantidad de publicaciones se comportaba de manera similar al precio a través del tiempo, ambos iban en aumento, le adjudicamos este comportamiento a la cantidad de publicaciones al hecho de que la empresa haya ganado popularidad a través de los años y el comportamiento del precio estamos bastante convencidos de que debe ser producto de la inflación.

Luego de probar esta alternativa y la anterior notamos que si las utilizábamos individualmente el score empeoraba pero al probarlas juntas el score mejoró bastante, superando incluso a One Hot Encoder, por lo que se conservaron ambos conjuntos de features y se descartaron así las más de 1000 columnas generadas por OneHotEncoder.

Baños

Luego del éxito de los features creados a partir de los targets sobre metros y cantidad de publicaciones nos pareció interesante seguir explotando este método. Retrocediendo en esta sección veíamos la importancia de los features numéricos originales, entre ellos teníamos la cantidad de baños presentes en las propiedades, notamos que la cantidad de baños estaba fuertemente relacionada al aumento del precio, observemos:

Precio promedio de propiedades según cantidad de baños



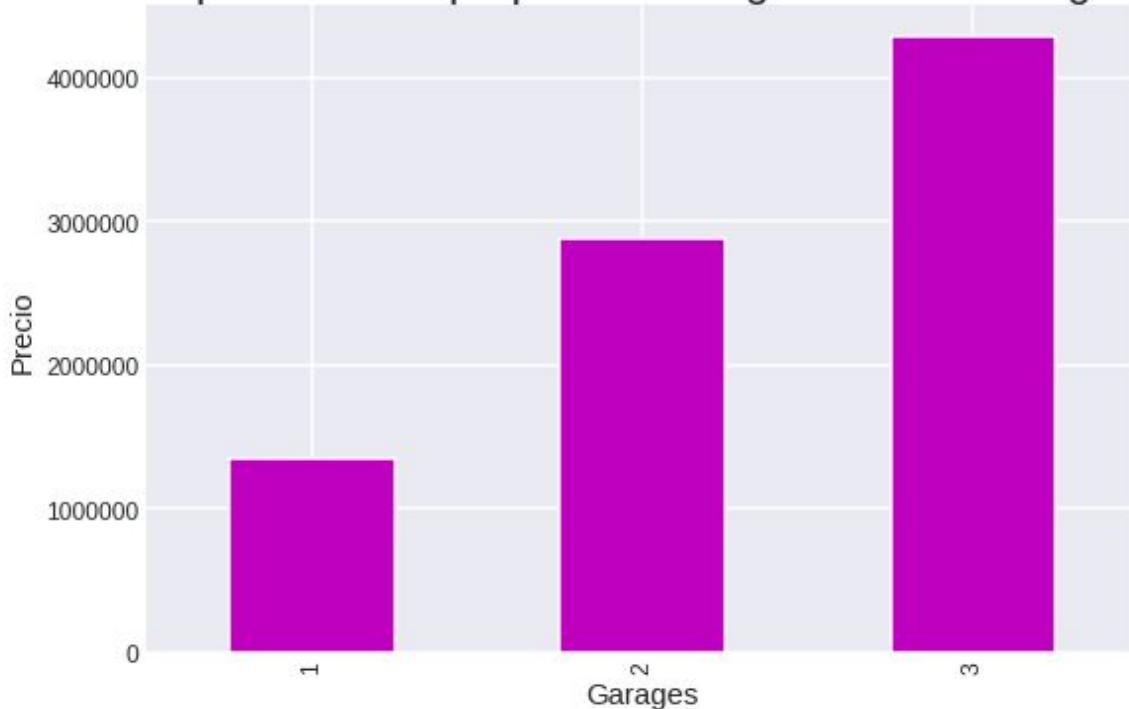
la relación está a simple vista, en promedio, cuanto más baños tenga una propiedad más elevado será su precio, por lo que decidimos aprovechar también ésta relación y crear un nuevo set de features utilizando este dato como target.

Y como esperábamos, estos features funcionaron muy bien y se encuentran en el set de features definitivo.

Garages

Otras de las cosas que observamos fue el comportamiento del precio de acuerdo a la cantidad de garajes que poseían las propiedades, veamos:

Precio promedio de propiedades según cantidad de garajes



Competencia de Machine Learning ZonaProp - Organización de Datos

Como se puede observar el precio va en aumento a medida que la cantidad de garages aumenta, como además este feature se hallaba entre los más importantes, tratamos de utilizar esta relación creando features mediante el uso de la cantidad de garages como target.

Este Feature también funcionó bastante bien, ayudando a mejorar el score por lo que fue conservado.

Habitaciones

Al igual que la cantidad de garages, la cantidad de habitaciones era uno de los features que aparecía con un peso considerable entre los features importantes, además recordábamos que durante el TP1 encontramos una cierta relación entre el precio y la cantidad de habitaciones.



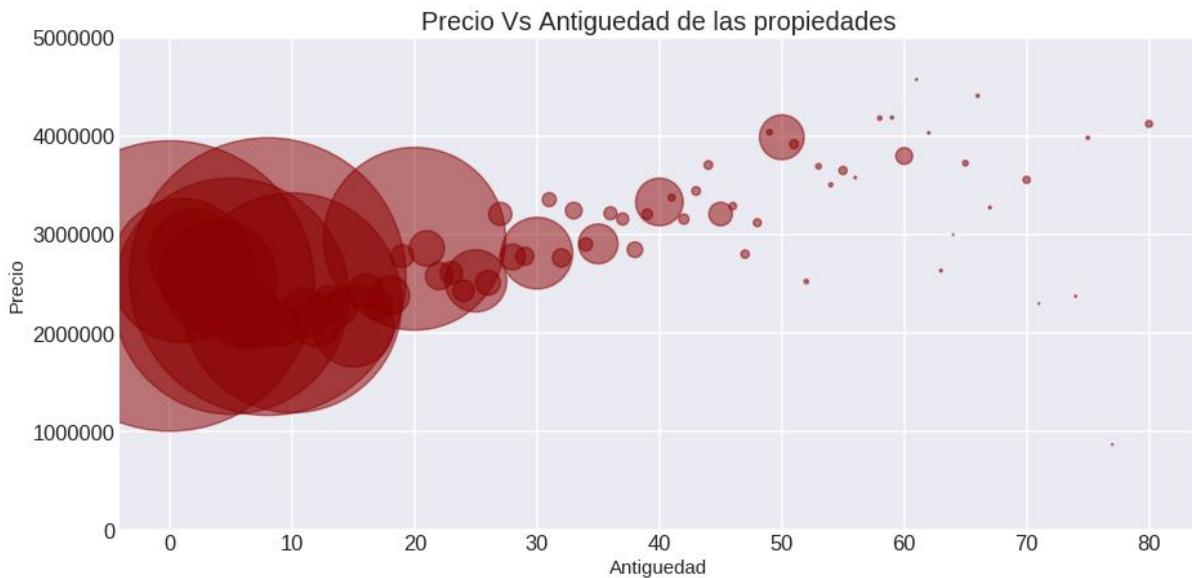
En el gráfico observamos como, en líneas generales el precio promedio va en aumento cuando aumenta la cantidad de habitaciones de una propiedad.

Aprovechando este datos procedimos de la misma manera que en los casos mencionados anteriormente, utilizando la cantidad de habitaciones como target. Los features generados por este método funcionaron muy bien, ayudando a disminuir el error en las predicciones de nuestro modelo, por lo que podrán encontrar estos features en el set de datos final.

Antigüedad

Este es uno de los datos que nos quedaba entre los features más importantes, incluso su importancia supera a la de habitaciones y garages. Para no sonar repetitivos, mencionaremos que el procedimiento para crear features basados en este dato fue exactamente el mismo que los mencionados anteriormente, dentro de target encoder.

Pero si nos interesa destacar la relación hallada entre el precio y la antigüedad.



Si bien no hay una relación muy clara entre la variación del precio y las propiedades menos antiguas, vemos que cuanto más antigua una propiedad el precio promedio es más elevado. Agregar este feature al set de features generó buenos resultados, haciendo que el score mejore, debido a esto podrán verlo entre los features utilizados en el modelo final.

Características: Gimnasio, Piscina, Usos Múltiples, Escuelas y Centros Comerciales Cercanos

Como lo dice el título estos atributos son los que consideramos como características particulares, estos datos vienen originalmente como columnas indicando con 1 o 0 si la propiedad posee alguna de éstas características, la idea era utilizar esta información de alguna manera, pero no sabíamos bien cómo hacerlo.

Para comenzar vimos que la importancia de estos features es la más baja dentro de los features originales, por lo que pensamos en combinarlos para crear un nuevo feature y así aumentar un poco su importancia. La manera que encontramos para hacer esto fue la siguiente, primero que nada creamos una nueva columna para indicar si existían propiedades que no contaran con ninguna de éstas características, llamada “no_possee”, de esta manera nos evitamos perder la información de propiedades que estuvieran en esa situación. El paso siguiente fue generar una nueva columna en donde aparecía, para cada propiedad (id), el nombre de la característica y luego otra columna que indica con 1 o 0 si la propiedad contaba con esa característica, llamada “si/no”. Como lo que necesitábamos era mantener el mismo número de filas que tenemos en el set de train y test respectivamente, luego se filtraron las propiedades que tuvieran 0 en la columna “si/no”, notemos que esto no nos hace perder información ni propiedades pues para propiedad teníamos 6 características posibles y nos quedamos sólo con la que tuviera un 1. Luego de esto notamos que había propiedades que contaban con más de una de éstas características por lo que se decidió concatenarlas para así tener una única fila para cada propiedad. Finalmente agregamos esta columna a target encoder para generar un nuevo feature numérico en base estas características. El resultado de agregar este feature al modelo no fue muy bueno, hizo que nuestro score subiera un poco, por lo que finalmente se decidió descartar al mismo.

Direcciones

Uno de los datos que aún no habíamos mencionado es el de las direcciones, un campo bastante variable, por decir algo, aunque a decir verdad cuenta con demasiados valores sin especificar, además como son direcciones, son muchos los valores que aparecen una única vez. Como no queríamos pasar por alto ninguna información decidimos probar algún feature sobre estos datos. Viendo que había muchos valores sin especificar o inválidos, como números, guione, puntos, símbolos sin sentido, etc. Tratamos de agrupar estos en un único tipo al cual llamamos “sin especificar”, luego de esto intentamos utilizar este como un atributo categórico más dentro de target encoder. Una vez obtenido el nuevo feature procedimos a probarlo, pero el resultado no fue bueno, lo cual era de esperarse, pues hay demasiada varianza en los valores y además no hay casi ninguna relación entre los precios y las direcciones, es más durante el desarrollo del TP1 vimos que no había demasiada diferencia entre precios de propiedades situadas en avenidas o calles. Finalmente este feature fue descartado también pues no aportaba nada positivo a nuestro modelo final.

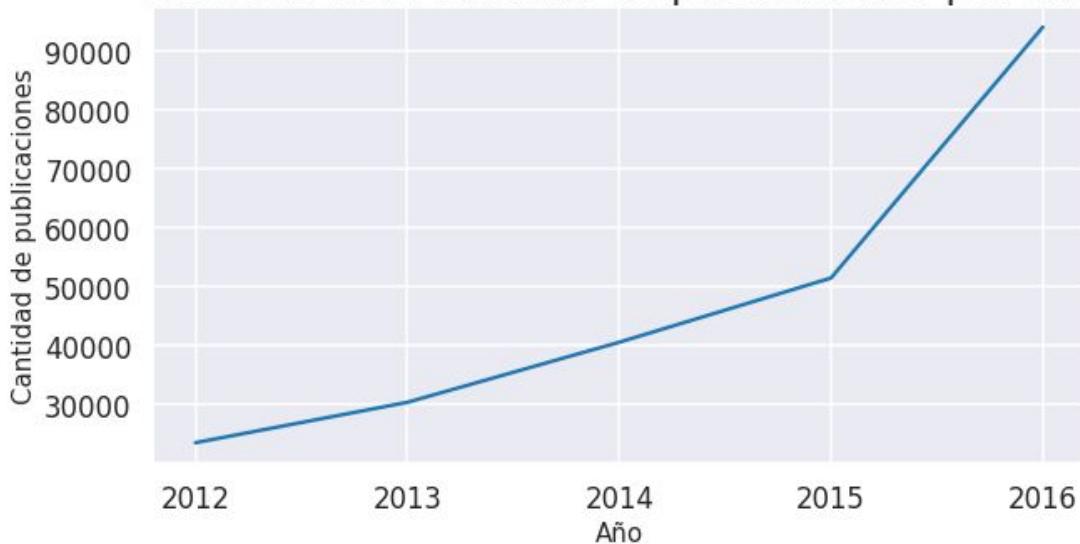
CatBoost Encoder

Otra de las alternativas que probamos cuando se intentaba reducir la cantidad de columnas producidas por One Hot Encoding fue CatBoost Encoder, el procedimiento fue bastante similar al de Target Encoder, por cada columna de valores categóricos se generaba una transformación de esos valores a un valor numérico basado en el target que le pasamos al encoder, lo intentamos de igual manera con metros cubiertos y cantidad de publicaciones anuales, pero este encoder no funcionó tan bien como lo hizo Target Encoder. Al retirar las columnas de One Hot Encoder y reemplazarlas por las generadas por este método el score de nuestro modelo empeoró bastante por lo que finalmente estos features se descartaron, quedando fuera del modelo final.

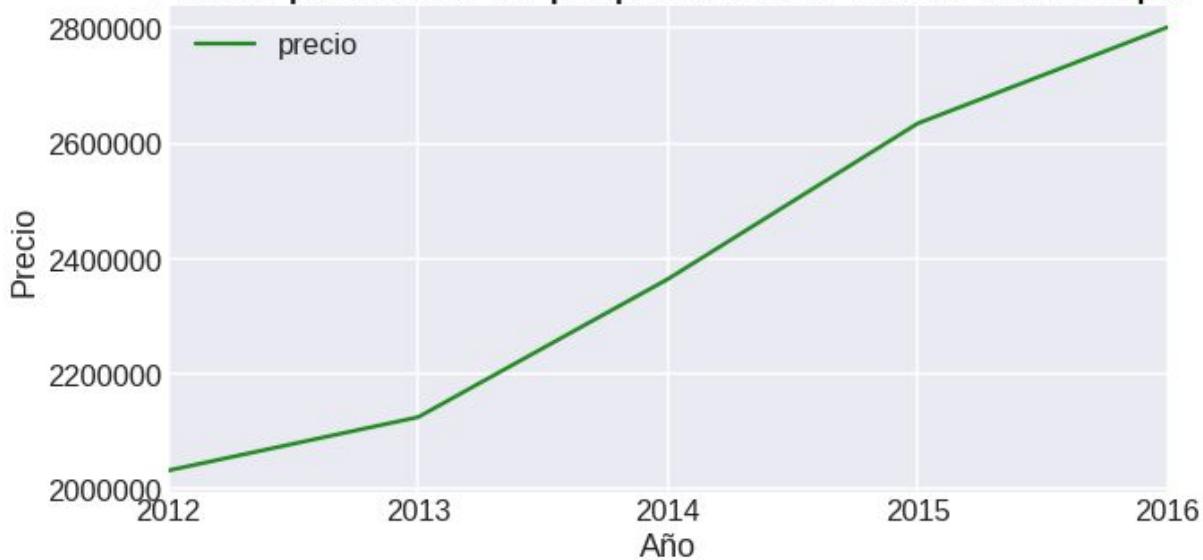
Features Sobre el tiempo

Durante el desarrollo del primer trabajo práctico descubrimos que había una fuerte relación entre el precio y el tiempo, así también como con la cantidad de publicaciones a través del tiempo. Ambos se comportan de manera similar, aumentan a medida que el tiempo aumenta. Debido a este descubrimiento nos pareció importante crear features relacionados con el tiempo. A continuación se puede apreciar esta relación en los gráficos.

Variación de la cantidad de publicaciones por año.



Precio promedio de propiedades a través del tiempo



Vemos que si bien existen ciertos codos el comportamiento en general es el mismo, van en aumento a través de los años.

“Puntaje por fecha”

Este feature se encuentra en el set de features como “puntaje_por_fecha” y fue el primer feature que se creó tomando en cuenta el tiempo en el que se realizó la publicación.

Para realizar este feature se realiza una suerte de “distancia”, en días, desde la fecha de publicación tomando en cuenta el tiempo de la primera publicación, tomando la fecha mínima, y el tiempo de la última publicación, tomando la fecha máxima, de la siguiente manera:

$$\text{puntaje_por_fecha} = \frac{\text{fecha_actual} - \text{fecha_mínima}}{\text{fecha_máxima} - \text{fecha_mínima}} \quad (*)$$

(*) La diferencia entre las fechas fue tomada en días.

Luego de probar nuestro modelo con este feature el score del mismo mejoró bastante, además luego de evaluar la importancia de los features mediante Random Forest este feature tiene una importancia bastante significativa, por lo que se encuentra actualmente entre los features del modelo final.

“Publicaciones por día / mes / año”

Como ya se mencionó al comienzo de esta sección la cantidad de publicaciones a través del tiempo se comporta de manera similar al precio, por lo que creamos diferentes features tomando en cuenta la cantidad de publicaciones realizadas para cada día, mes y año, los mismos se pueden encontrar nombrados como “publicaciones_por_dia”, “publicaciones_por_mes” y “publicaciones_por_año”. Se probó agregar cada feature por separado y luego en conjunto evaluando el score y registrando la variación del mismo con cada cambio, el feature que finalmente nos arrojó mejores resultados luego de agregarlo fue “publicaciones_por_año” y se lo puede encontrar al mismo entre los features del set final.

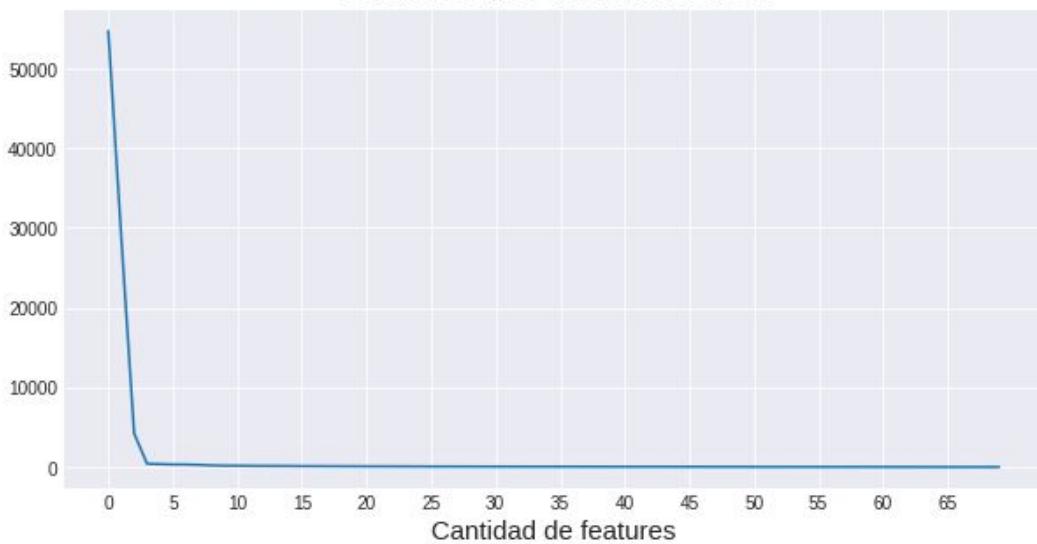
Features creados a partir del uso de SVD

Nuevos features a partir de la reducción del set de features

Inicialmente nos pareció interesante probar el uso de la SVD como método de feature engineering, reduciendo nuestros datos originales a unas pocas columnas y utilizar estas como nuevos features. Al momento de probar este método contábamos con 70 columnas, lo que hicimos fue aplicar la SVD sobre estas columnas y ver cuál era la dimensionalidad intrínseca de estos datos.

En el siguiente gráfico se puede observar como hay un notorio codo entre 0 y 5, esto indicaría que con menos de 5 dimensiones podemos capturar casi la totalidad la información de nuestro set de features.

Gráfico de autovalores



Para poder determinar certeramente la cantidad de dimensiones a utilizar realizamos el cálculo de la energía y evaluamos cuánta energía se conserva de acuerdo a la cantidad de autovalores que se van tomando, es decir, de acuerdo a la cantidad de dimensiones. Esto se puede observar a continuación:

- ⇒ Cantidad de valores singulares 1: 54656.96594186719 77.7%
- Cantidad de valores singulares 2: 28955.255696048487 99.51%
- Cantidad de valores singulares 3: 4225.706334713612 99.98%
- Cantidad de valores singulares 4: 411.48632369035244 99.98%
- Cantidad de valores singulares 5: 364.9108791838578 99.98%
- Cantidad de valores singulares 6: 325.0991308335752 99.99%
- Cantidad de valores singulares 7: 314.38873055803134 99.99%
- Cantidad de valores singulares 8: 270.69362094590707 99.99%
- Cantidad de valores singulares 9: 203.8729050488148 99.99%
- Cantidad de valores singulares 10: 172.2051258890939 99.99%
- Cantidad de valores singulares 11: 162.40172352795452 99.99%
- Cantidad de valores singulares 12: 146.37380461820118 99.99%
- Cantidad de valores singulares 13: 136.74258285705804 100.0%
- Cantidad de valores singulares 14: 134.34419666898484 100.0%
- Cantidad de valores singulares 15: 125.1939121283043 100.0%

Como habíamos mencionado, con 3 dimensiones tenemos el 99.98% de la información, y si tomamos 13 dimensiones capturamos el 100%. Con esta información decidimos crear 3 nuevos features basándonos en la reducción de nuestros features a 3 dimensiones.

Al agregar estos features al set de features el resultado no fue muy bueno, de hecho empeoró bastante, esto tal vez se debe al hecho de estar repitiendo información, pues se trata de una versión reducida de los features totales, debido al mal resultado los features fueron finalmente descartados.

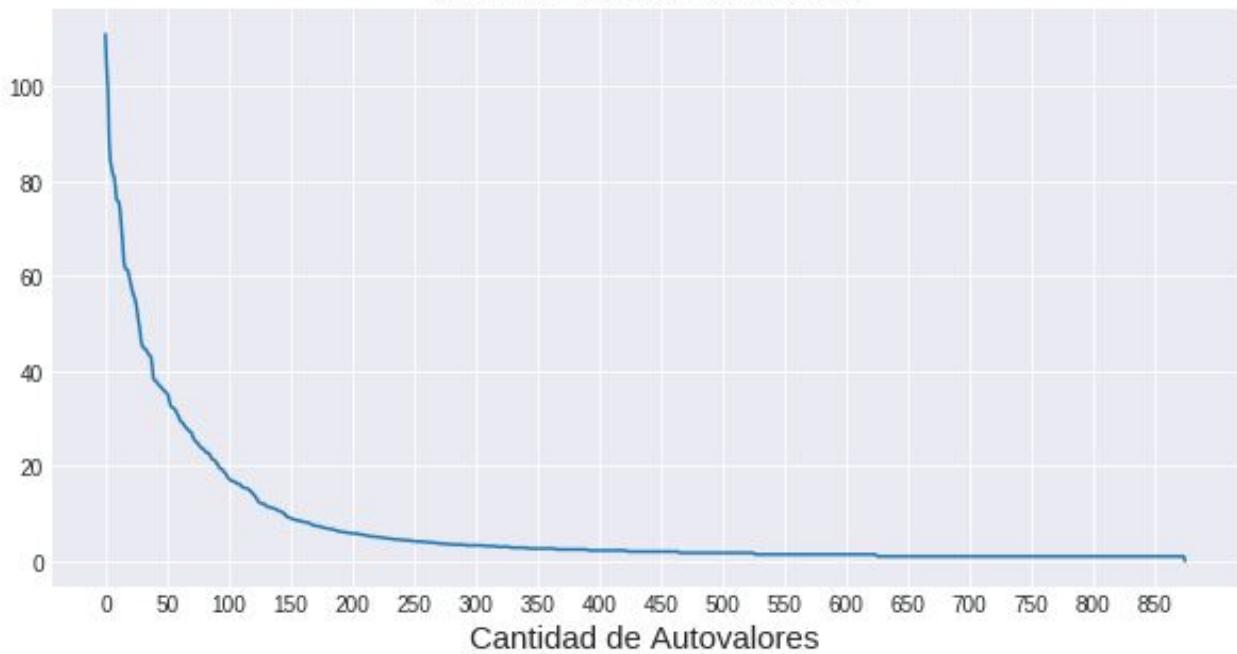
SVD sobre One Hot Encoding

Ciudades

Como ya hemos mencionado en una instancia contábamos con más de 1000 features, en ese entonces habíamos creado features con One Hot encoding a partir de las provincias y tipos de propiedad de las publicaciones, pero aún nos faltaba utilizar la información a cerca de las ciudades. Analizando la cantidad de ciudades notamos que en el set de train contábamos con más de 800 ciudades, por lo que no era una opción agregar todas esas columnas al set de features, serían casi 2000 columnas y de por sí nuestro modelo se habría vuelto muy lento con las ya 1000 columnas que contaba.

En esa instancia decidimos probar nuevamente la SVD. Lo que hicimos fue aplicar One Hot Encoding generando una columna para cada ciudad existente, indicando 1 o 0 según si la propiedad se sitúa o no en esa ciudad, luego aplicamos la SVD sobre este set. Lo primero que hicimos fue graficar los autovalores y detectar los codos.

Gráfico de autovalores



Como se puede ver en el gráfico con unas 80 dimensiones se puede capturar una gran parte de la información de nuestro set de datos, y tomando unas 200 dimensiones capturamos casi la totalidad de la información, para poder observar mejor esto realizamos el cálculo de la energía acumulada de acuerdo a la cantidad de autovalores conservados:

Competencia de Machine Learning ZonaProp - Organización de Datos

```
[89] Cantida de Valores Singulares 73: 13.415633959129075 88.13%
     Cantida de Valores Singulares 74: 13.19699151684704 88.42%
     ↳ Cantida de Valores Singulares 75: 13.066397488527782 88.71%
     Cantida de Valores Singulares 76: 13.000000000000018 89.0%
     Cantida de Valores Singulares 77: 12.972205717846041 89.29%
     Cantida de Valores Singulares 78: 12.797199752821706 89.57%
     Cantida de Valores Singulares 79: 12.408302359256771 89.83%
     Cantida de Valores Singulares 80: 12.305034047290432 90.08%
     Cantida de Valores Singulares 81: 11.910432528840705 90.33%
     Cantida de Valores Singulares 82: 11.746652923964911 90.56%
     Cantida de Valores Singulares 83: 11.474494241314034 90.78%
     Cantida de Valores Singulares 84: 11.284093207717827 91.0%
     Cantida de Valores Singulares 85: 11.107311397558842 91.21%
     Cantida de Valores Singulares 86: 10.96429050365726 91.42%
     Cantida de Valores Singulares 87: 10.881396964608303 91.62%
     Cantida de Valores Singulares 88: 10.71703330913118 91.81%
     Cantida de Valores Singulares 89: 10.4956864717326 92.0%
     Cantida de Valores Singulares 90: 10.316439984255389 92.18%
     Cantida de Valores Singulares 91: 10.220031542952036 92.36%
     Cantida de Valores Singulares 92: 10.164940470483828 92.53%
     Cantida de Valores Singulares 93: 9.757571703987574 92.7%
     Cantida de Valores Singulares 94: 9.543365229613162 92.85%
     Cantida de Valores Singulares 95: 9.30706663594351 93.0%
     Cantida de Valores Singulares 96: 9.160752641373632 93.14%
     Cantida de Valores Singulares 97: 9.07762826041183 93.28%
     Cantida de Valores Singulares 98: 8.931581434958613 93.42%
     Cantida de Valores Singulares 99: 8.766687956580313 93.55%
     Cantida de Valores Singulares 100: 8.571848586538664 93.67%
```

Como se puede observar con 80 dimensiones se captura el 90% de la energía de set de datos, por lo que decidimos aplicar esa reducción.

Ya aplicada la reducción a 80 dimensiones agregamos esos 80 features al set de features y corrimos el modelo para ver cómo funcionaban nuestros nuevos features, lamentablemente el score final no se vio beneficiado, empeorando nuevamente, por lo que también se decidió descartar estos features.

Provincias y tipos de propiedad

En varias de las secciones anteriores mencionamos que contábamos con más de 1000 features que eran producto de aplicar One Hot Encoding a variables categóricas como las provincias y tipo de propiedad, este número de features nos limitó bastante a la hora de probar modelos y nuevos features, pues el tiempo que tardamos en entrenar se había incrementado demasiado.

El trabajo de encontrar la mejor manera de reducir el set de features sin perder la precisión de nuestras predicciones se volvió bastante arduo y entre los tantos métodos que probamos se encontró nuevamente la SVD.

Realizamos el mismo método ya mencionado, separamos nuestros mas de 1000 features en un set aparte y aplicamos la SVD sobre este set. Luego analizamos los autovalores en un gráfico y decidimos aplicar una reducción de acuerdo a lo observado en el comportamiento según la cantidad de autovalores.



Vemos que con unas 200 dimensiones, aproximadamente, capturamos la mayor parte de la información de nuestro set de datos y si tomáramos unas 500 dimensiones capturaremos casi la totalidad de la información.

Decidimos realizar una reducción de 200 dimensiones y agregar estas columnas como nuevos features a nuestro set de datos, luego de esto probamos el modelo y el score aumentó bastante, supusimos que esto podría deberse a la pérdida de información, por lo que decidimos probar con 500 dimensiones, recordando que inicialmente contábamos con más de 1000, el resultado mejoró un poco pero no lo suficiente, por lo que seguía siendo más conveniente mantener las más de 1000 columnas en lugar de sacrificar la precisión de nuestras predicciones.

Features sobre el Precio (precio_min y precio_max)

Como la competencia tiene que ver con predecir el precio de las propiedades nos pareció interesante utilizar la información sobre los precios de las propiedades que nos brindaban en el set de train. A continuación detallamos cómo utilizamos esta información para crear nuevos features.

Rango de precio esperado según la ciudad

La idea detrás de este feature era crear un rango de valores para los precios que pueden tomar las propiedades según las ciudades en las que se encuentran. Para esto se tomó el valor mínimo y máximo del precio para cada ciudad y se generaron dos nuevas columnas con estos valores, asignándose de acuerdo a la ciudad en la que se encuentra cada propiedad.

Al momento de asignar los máximos y los mínimos notamos que había ciudades que tenían un mismo valor para el mínimo y el máximo, creemos que esto puede deberse a que sólo haya una publicación en esa ciudad, o las publicaciones correspondan a la misma propiedad o algún conjunto de propiedades que correspondan a un conjunto de propiedades del mismo tipo. Nuestra solución para esos casos fue crear un rango mucho más amplio y para hacerlo tomamos el mínimo de todas

las ciudades y el máximo de todas las ciudades, de esta manera ampliamos el rango de las ciudades con poca varianza en los precios. Para poder agregar estos features al set de test lo que hicimos fue usar un join de acuerdo a las ciudades, luego de esto notamos que había ciudades nuevas en el set de test, por lo que procedimos a ampliar el rango de la misma manera que lo habíamos hecho para las ciudades donde sólo teníamos un único valor.

Luego de agregar estos features al set de features y probarlos el resultado fue muy malo, llegando a duplicar el valor del error que teníamos hasta el momento, por lo que fueron descartados.

Rango de precios esperado según provincia y fecha de la publicación

Luego del fracaso de los features anteriores pensamos que esto podría deberse al hecho de que teníamos muchas ciudades y la varianza entre los precios no era lo suficientemente “alta” para estimar un rango parecido al real, y además había ciudades que no estaban presentes en el set de train pero si en el set de test o viceversa. Con esto en mente se nos ocurrió tomar en cuenta otro tipo de cosas como las provincias, las cuales son mucho menos, lo cual implica mayor cantidad de publicaciones por provincias y como además ya sabíamos sobre la influencia del tiempo en el precio, decidimos tomar un rango según la provincia y la fecha, el procedimiento fue exactamente el mismo que el caso anterior pero tomando en cuenta estos dos atributos. Una vez creado nuestro rango procedimos a probarlo, si bien el resultado ya no fue tan malo como el caso anterior, igualmente no funcionó muy bien, aumentando el score que teníamos hasta ese momento, por lo que fue descartado del modelo definitivo.

Features Sobre Palabras en Títulos y Descripciones

Originalmente en el TP1 habíamos tratado de encontrar palabras en títulos y descripciones que estén correlacionadas a un menor o mayor precio de la publicación que las contenía. Las palabras que habíamos encontrado fueron las siguientes:

Palabras importantes:

En título:

Palabras correlacionadas a mayor precio:

'polanco','loma','herradura','condesa','bosque','country','hermosa','vista','residencia','condominio','golf','lujo','huixquilucan'

Palabras correlacionadas a menor precio:

'terreno','fracc','casa','villa','remate','recamaras','ecatepec','cerca','coacalco','izcalli','planta','lote','bonita','cautitlan'

En descripción:

Palabras correlacionadas a mayor precio:

'cuarto','vestidor','terraza','family','vista','bodega',
'salón','estudio','jardín','room','jacuzzi','lujo','antecomedor',
'tv','fiestas','juegos','estacionamiento','gimnasio','nado',
'lugares','doble','desayunador','servicio','chimenea','visita','acabado',
'espacio','salon','alberca','garden','spa'

Palabras correlacionadas a menor precio:

'minutos','acept','boiler','reja','credito','transporte','cochera','contado',
'escuela','inf','fovissste','cerca','bancario','protecciones','patio','infonavit'

Teniendo estas palabras en cuenta se nos ocurrió hacer, para cada publicación, una columna indicando, por cada palabra, si esa palabra aparece o no en la respectiva descripción o el respectivo título de esa publicación. Por ejemplo, como la palabra 'salón' para nosotros es importante en la descripción, creamos una columna llamada 'desc_salon' que para cada publicación va a valer 1 si esa publicación tenía esa palabra en su descripción y 0 en caso contrario.

Este método lo utilizamos casi desde el principio de la competencia logrando una mejora en el score en kaggle.

Intentamos tratar de encontrar otras palabras/métodos para explotar los datos de descripciones y títulos pero no hubieron mejores resultados que con el método anterior.

Se intentó:

- En vez de lo anterior usar The Hashing Trick tanto para título como para descripción, se probó con hasta 500 columnas para las descripciones pero los resultados no eran buenos comparados al método anterior así que se lo descartó.
- Sacar de la lista de palabras importantes a las palabras que, luego de entrenar con random forest con todos los features totales, tenían menos importancia para el modelo. Palabras como, para las de descripción, 'lujo', 'acept' o 'infonavit' entre otras terminaban con muy poca importancia. Se probó sacar todas esas palabras pero solo empeoraba el score.
- Se probó un nuevo modelo de random forest, se utilizó la librería de procesamiento de lenguaje natural **nltk** para contabilizar todas las palabras (por separado para descripción y para título), ignorando las stopwords del lenguaje español (esto es una funcionalidad que brinda nltk), quedarnos con 200 palabras que más apariciones tuvieron en título y 300 que más apariciones tuvieron en descripción, hacer el mismo procedimiento de por cada palabra una columna con 1 o 0 indicando si la publicación tenía esa palabra o no. Luego se entrenó el modelo de random forest solo con esas columnas y nos quedamos con las que más importancia tuvieron para el modelo. Sin embargo muchas de las palabras coincidían con las que ya habíamos encontrado y, además, utilizar estas nuevas palabras empeoró el score, también lo empeoró hacer una unión de las que ya teníamos con estas nuevas, así que esto simplemente fue ignorado y nos quedamos con las que ya teníamos. Algunas de las palabras que habían sido importantes para el título según este enfoque eran: venta, casa,

Competencia de Machine Learning ZonaProp - Organización de Datos

edificio, departamento, terreno, condesa. Y para descripción fueron: cuarto, patio, vestidor, family, infonavit, uso,... entre otras.

Modelos de Machine Learning

Modelos Baseline

Para comenzar decidimos probar 3 modelos como baseline, de esta manera tendríamos un umbral a partir del cual comenzar a evaluar nuestros features.

Los modelos se evaluaron calculando en score con el uso de la métrica MAE mediante el uso de Cross Validation.

Linear Regression

Como primer aproximación optamos por una Regresión Lineal, este nos indica el punto de partida para nuestro score. Se utilizó el algoritmo sin modificar ningún hiper parámetro, pues no se creyó necesario ya que esperábamos que este sea el peor resultado y luego de allí comenzaría a subir a medida que fuésemos probando los diferentes modelos.

A continuación se puede observar cuál fue el resultado de este modelo:

```
▶ #Score MAE
lr_model = LinearRegression()
scores = cross_val_score(lr_model , X, Y, scoring="neg_mean_absolute_error", cv=3)
RF_mae_score1 = -scores.mean()
RF_mae_score1
936327.020686386
```

KNN Regressor

Como segunda opinión probamos con KNN, la idea era comparar la diferencia entre este modelo y el anterior, particularmente esperábamos que nos de mejor a una regresión lineal. Sin necesidad de mejorar hiper parámetros, corrimos el modelo y obtuvimos el siguiente resultado:

```
[ ] knn_model = KNeighborsRegressor()

[ ] #Score MAE
scores = cross_val_score(knn_model , X, Y, scoring="neg_mean_absolute_error", cv=3)
RF_mae_score3 = -scores.mean()
RF_mae_score3
847298.52512
```

Vemos que el resultado fue mucho mejor que el modelo anterior por lo que pensamos que valdría la pena realizar una búsqueda de hiper parámetros para lograr optimizar este modelo. Esto lo realizamos a través de Grid Search, luego de investigar un poco dimos con un rango de valores los cuales fuimos evaluando a través de este método, el resultado fue el siguiente:

Competencia de Machine Learning ZonaProp - Organización de Datos

```
parameters_for_testing = {'metric': ['euclidean', 'manhattan', 'chebyshev', 'minkowski'], \
                           'weights': ['uniform', 'distance'], 'n_neighbors': list(range(1,15, 3))} \
gsearch1 = GridSearchCV(estimator = knn_model, param_grid = parameters_for_testing,iid=False, verbose=10,scoring='neg_mean_absolute_error', n_jobs = 2) \
gsearch1.fit(X,Y) \
print('best params') \
print (gsearch1.best_params_) \
print('best score') \
print (-gsearch1.best_score_) \
best params \
{'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'distance'} \
best score \
747513.4815954762
```

Como se puede ver la búsqueda de hiper parámetros fue bastante beneficiosa, logrando una gran mejora en el score final.

Además de Grid Search también probamos Randomized Search, aumentando la cantidad de vecinos a probar, pero los resultados no fueron tan buenos, por lo que se conservaron los hiper parámetros obtenidos mediante Grid Search.

Más adelante detallaremos cómo utilizamos este modelo en un ensamble intentando así mejorar nuestras predicciones.

De forma adicional, para poder mejorar este modelo y usarlo en un ensamble como se mencionó, se probaron distintas técnicas que finalmente se descartaron porque no resultaron en ninguna mejora.

Estas se listan a continuación:

- Pruebas con features esenciales: Se probó el modelo con solo cuatro features que a priori podían ser los más importantes: metros cubiertos, metros totales, tipo de propiedad y puntaje por fecha. La idea de hacer esto era que como knn funciona mejor con pocas dimensiones, tal vez solo con estos podía haber una pequeña mejora. Sin embargo no funcionó porque para calcular el precio final evidentemente hay que tener en cuenta más factores.
- Usar solo propiedades anteriores en fecha: Como los precios van en aumento a lo largo del tiempo, tal vez era mejor comparar solamente con registros previos para evitar predecir los precios con valores futuros y evitar problemas de *time traveling*, pero no hubo mejora, tal vez porque la cantidad de registros contra los cuales comparar a veces se hacía muy pequeño.
- Usar solo propiedades dentro de los dos meses anteriores y los dos posteriores: La razón para esto es parecida a la anterior, pero como aquí se usan los meses más cercanos los precios deben ser lo más parecidos posibles. Pero este enfoque también reduce mucho la cantidad contra la que comparar y degrada la predicción.
- Utilizar solo los registros sin valores nulos: La idea aquí es usar valores sin el ruido introducido al llenar los nulos y que sean valores más “puros”, y efectivamente las predicciones mejoran para los valores sin nulos, pero para los que sí tienen faltantes la predicción se vuelve peor, por lo que se descarta ya que en el test también hay muchos faltantes.
- Creación de clusters por fecha: Aquí se dividieron los datos por fechas en grupos de partes iguales por cercanía del tiempo de publicación y se calculó el centroide de cada grupo, para luego comparar el registro a predecir contra cada centroide y quedarse con el más cercano. Por último se utilizaron los vecinos más cercanos dentro de ese grupo para la predicción final. El objetivo era agilizar la búsqueda para las propiedades más cercanas en tiempo, pero el resultado no fue bueno.
- Creación de clusters por rango de precios: En este caso nuevamente se crearon clusters pero para rangos de precios similares, con la esperanza de que el centroide más parecido al objetivo fuera de características similares, pero seguramente debido a la diversidad de propiedades en rangos similares de precios, no resultó.

- Selección de rangos de variables: Este fue el último enfoque probado, y lo que se hizo fue reducir la cantidad de valores contra los cuales comparar filtrando las propiedades para que quedaran las que estuvieran en un rango cercano con respecto a algunos features como metros cubiertos, habitaciones y puntaje por fecha. El problema fue que al ir filtrando sucesivamente, quedaban cada vez menos contra los cuales comparar, muchas veces menos de la cantidad de vecinos para los cuales estaba programado nuestro knn y seguramente por eso no dió tan buenos resultados.

Decision Tree Regressor

Como último modelo baseline probamos con un árbol de decisión. Probamos el modelo utilizando los hiper parámetros por default y el resultado fue el que se puede observar a continuación:

```
[ ] dtr_model = DecisionTreeRegressor()  
  
[ ] #Score MAE  
scores = cross_val_score(dtr_model , X, Y, scoring="neg_mean_absolute_error", cv=2)  
RF_mae_score2 = -scores.mean()  
RF_mae_score2  
  
👤 800097.5959679165
```

Como se puede observar el resultado inicial es incluso mejor que KNN, es por esta razón que decidimos realizar una búsqueda de hiper parámetros y así ver hasta dónde podíamos mejorar nuestro score. Para esto probamos utilizando Grid Search, no sin antes investigar un poco acerca de los rangos de valores que podríamos pasar a cada hiper parámetro.

```
[ ] parameters_for_testing = {'max_depth':[14,15,16,17,18,19,20] , 'min_samples_split': [14,15,16,17,18,19,20],\n                             'max_features': [92,93,94,95,96,97,98]}  
  
gsearch1 = GridSearchCV(dtr_model, param_grid=parameters_for_testing, cv=2, iid=False, scoring='neg_mean_absolute_error', n_jobs = 2)  
gsearch1.fit(X,Y)  
print('best params')  
print (gsearch1.best_params_)  
print('best score')  
print (-gsearch1.best_score_)  
  
👤 best params  
{'max_depth': 16, 'max_features': 95, 'min_samples_split': 20}  
best score  
711948.7258354134
```

Luego de varias pruebas, tanto con Grid Search y Randomized Search, llegamos al mejor resultado con los hiper parámetros que se pueden observar en la imagen.

Más adelante veremos cómo este modelo será considerado para realizar ensambles junto con otros modelos.

Random Forest

Este fue el primer modelo que utilizamos para evaluar la evolución de nuestros features, ya sea calculando el error mediante Cross Validation como también para medir la importancia de cada uno de nuestros nuevos features, y también crear las predicciones que luego subímos a Kaggle.

Competencia de Machine Learning ZonaProp - Organización de Datos

Este modelo nos funcionó tan bien desde el comienzo que estuvimos convencido durante bastante tiempo de que sería nuestro modelo definitivo, por lo que invertimos bastante tiempo en buscar los hiper parámetros que mejor se ajustaran a nuestro set de features, esto lo hicimos en un principio a través de Grid Search pero luego, para poder explorar entre más valores posibles, comenzamos a probar con Randomized Search.

Observemos primero cuál era el score de este modelo sólo con los hiper parámetros definidos por default:

```
rf_model = RandomForestRegressor(oob_score= True)
scores = cross_val_score(rf_model , X, Y, scoring="neg_mean_absolute_error", cv=2)
RF_mae_score5 = -scores.mean()
RF_mae_score5

/home/sol/.local/lib/python3.6/site-packages/sklearn/ensemble/forest.py:245: FutureWarning: The default '10 in version 0.20 to 100 in 0.22.", FutureWarning)
/home/sol/.local/lib/python3.6/site-packages/sklearn/ensemble/forest.py:737: UserWarning: Some inputs do warn("Some inputs do not have OOB scores. "
/home/sol/.local/lib/python3.6/site-packages/sklearn/ensemble/forest.py:245: FutureWarning: The default '10 in version 0.20 to 100 in 0.22.", FutureWarning)
/home/sol/.local/lib/python3.6/site-packages/sklearn/ensemble/forest.py:737: UserWarning: Some inputs do warn("Some inputs do not have OOB scores. "
595248.649810446
```

Como se puede observar el resultado es bastante bueno, en comparación con los vistos anteriormente. Luego de múltiples pruebas llegamos al siguiente resultado:

```
rf_model = RandomForestRegressor(random_state = 63, n_estimators = 100, min_samples_split = 5, min_samples_leaf = 1,
                                 max_features = 101, max_depth = 40, oob_score = True)
scores = cross_val_score(rf_model , X, Y, scoring="neg_mean_absolute_error", cv=2)
RF_mae_score = -scores.mean()
RF_mae_score

530595.1908701155
```

Como se puede apreciar el error disminuyó bastante. Estos son los hiper parámetros que usamos para el modelo final de Random Forest utilizado.

Gradient Boosting

Comentaremos brevemente como fue nuestro paso por este modelo. Inicialmente comenzamos probandolo con los hiper parámetros por default, como con el resto de los modelos, el resultado fue el siguiente:

```
[7] gbr_model = GradientBoostingRegressor()

[8] scores = cross_val_score(gbr_model , X, Y, scoring="neg_mean_absolute_error", cv=3)
RF_mae_score6 = -scores.mean()
RF_mae_score6

703552.3457100395
```

Vemos que es realmente muy malo si lo comparamos con Random Forest, pero aún le teníamos algo de fé, así que decidimos realizar una búsqueda de hiper parámetros y ver cuanto podíamos hacer que mejore. El mejor resultado obtenido, luego de varias pruebas, fue el que pueden observar a continuación:

Competencia de Machine Learning ZonaProp - Organización de Datos

```
RandomizedSearchCV(cv=3, error_score='raise-deprecating',
                    estimator=GradientBoostingRegressor(alpha=0.9,
                                                        criterion='friedman_mse',
                                                        init=None,
                                                        learning_rate=0.1,
                                                        loss='ls', max_depth=3,
                                                        max_features=None,
                                                        max_leaf_nodes=None,
                                                        min_impurity_decrease=0.0,
                                                        min_impurity_split=None,
                                                        min_samples_leaf=1,
                                                        min_samples_split=2,
                                                        min_weight_fraction_leaf=0.0,
                                                        n_estimators=100,...,
                                                        9, 10, 11, 12, 13, 14,
                                                        15, 16, 17, 18, 19, 20,
                                                        21, 22, 23, 24, 25, 26,
                                                        27, 28, 29, 30, ...],
                    'min_samples_leaf': array([0.1, 0.2, 0.3, 0.4, 0.5]),
                    'min_samples_split': array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
                    'n_estimators': [1, 2, 4, 8, 16, 32, 64,
                                    100, 200]},
                    pre_dispatch='2*n_jobs', random_state=None, refit=True,
                    return_train_score=False, scoring='neg_mean_absolute_error',
                    verbose=0)

print('best params')
print (random_search.best_params_)
print('best score')
print (-random_search.best_score_)

best params
{'n_estimators': 100, 'min_samples_split': 0.1, 'min_samples_leaf': 0.1, 'max_features': 86, 'max_depth': 25.0, 'learning_rate': 0.5}
best score
708438.0012370601
```

Como se puede ver el resultado es muy malo, incluso peor que los parámetros por default, por esta razón este modelo no fue considerado dentro del modelo final.

XGBoost

Este modelo es el modelo principal dentro de nuestro modelo final utilizado para generar las predicciones, es uno de los modelos a los cuales le dedicamos más tiempo, y en el cual buscamos hiper parámetros de manera más exhaustiva, pues a medida que probamos veíamos que siempre podía mejorar.

En un principio hicimos la prueba como con el resto de los modelos, sin hiper parámetros definidos más que los que vienen por defecto, el resultado fue el siguiente:

```
[9] xgb_model = xgb.XGBRegressor()

[10] scores = cross_val_score(xgb_model , X, Y, scoring="neg_mean_absolute_error", cv=3)
    RF_mae_score7 = -scores.mean()
    RF_mae_score7

(User warning: /usr/local/lib/python3.6/dist-packages/xgboost/core.py:587: FutureWarning: Series.base is deprecated at version 0.12 and will be removed at version 0.14. If you are using this feature, please use copy() to make a copy of the data before changing the base data frame. This warning is shown once per session.)
[14:51:28] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in fa
/usr/local/lib/python3.6/dist-packages/xgboost/core.py:587: FutureWarning: Series.base is deprecated at
    if getattr(data, 'base', None) is not None and \
[14:52:14] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in fa
/usr/local/lib/python3.6/dist-packages/xgboost/core.py:587: FutureWarning: Series.base is deprecated at
    if getattr(data, 'base', None) is not None and \
[14:53:00] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in fa
702816.654511991
```

Competencia de Machine Learning ZonaProp - Organización de Datos

Como se puede ver inicialmente el resultado es bastante malo, similar al de Gradient Boosting, luego de realizar una cuantas búsquedas de hiper parámetros, mediante Randomized Search, llegamos resultados realmente muy buenos, incluso superando a Random Forest!

```
xgb_model = xgb.XGBRegressor(subsample = 0.8, reg_alpha = 1, random_state = 61, n_estimators = 200, min_child_weight = 1, \
                               max_features = 105, max_depth = 9, learning_rate = 0.1, gamma = 0.1, colsample_bytree = 0.9)
scores = cross_val_score(xgb_model, X, Y, scoring="neg_mean_absolute_error", cv=2)
RF_mae_score = -scores.mean()
RF_mae_score

/usr/local/lib/python3.6/dist-packages/xgboost/core.py:587: FutureWarning: Series.base is deprecated and will be removed in a fut
  if getattr(data, 'base', None) is not None and \
[18:29:31] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
/usr/local/lib/python3.6/dist-packages/xgboost/core.py:587: FutureWarning: Series.base is deprecated and will be removed in a fut
  if getattr(data, 'base', None) is not None and \
[18:33:19] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
515887.6296143199
```

Debido a estos resultados es que este modelo es uno de los más importantes dentro del modelo final.

A continuación detallaremos cómo utilizamos este modelo dentro de varios ensambles buscando de esta manera mejorar aún más sus resultados.

AdaBoost Regressor

Luego de obtener los mejores resultados posibles para nuestros modelos por separado decidimos comenzar a probar ensambles entre ellos. Uno de los ensambles que probamos fue AdaBoost, a este modelo podemos pasarle un modelo por parámetro para que lo use como estimador, la única condición que debe cumplir nuestro modelo es que debe ser un árbol, lo cual era una ventaja para nosotros pues nuestros mejores modelos están basados en árboles.

Comenzamos pasándole como estimador un árbol de decisión, y probamos el modelo para ver si generaba una mejora de acuerdo al modelo base por sí sólo, el resultado fue el siguiente:

```
dtr_model = DecisionTreeRegressor(max_depth = 16, max_features = 95, min_samples_split = 20)
abr_model = AdaBoostRegressor(base_estimator=dtr_model)

scores = cross_val_score(abr_model, X, Y, scoring="neg_mean_absolute_error", cv=2)
abr_mae_score = -scores.mean()
abr_mae_score

626164.513684944
```

Como se puede ver le pasamos el árbol de decisión con los mejores hiper parámetros que habíamos encontrado antes para ese modelo, recordemos que el árbol por sí sólo nos arrojó un score en el orden de los 710000, por lo que, utilizarlo dentro de Adaboost hizo que el modelo mejorará!

Luego de ver este resultado no podíamos no probar con nuestros modelos estrellas, primero probamos pasarle Random Forest como estimador y veamos el resultado:

```
abr_model = AdaBoostRegressor(base_estimator=rf_model, n_estimators = 10)

scores = cross_val_score(abr_model , X, Y, scoring="neg_mean_absolute_error", cv=2)
abr_mae_score = -scores.mean()
abr_mae_score
```

531977.282683154

Al igual que antes le pasamos el modelo con los hiper parámetros ya modificados en la mejor versión hallada, en este caso el uso de Adaboost no hizo que el modelo mejore, recordemos que el mejor resultado de Random Forest fue de 530595.1908701155, por lo que procedimos a probar con otro modelo.

Probamos entonces utilizando XGBoost como estimador para Adaboost y el resultado fue un score de 507040.9576514893 el cual es mejor que el de XGBoost en solitario, el mismo se logró con los siguientes hiper parámetros.

```
xgb_model = xgb.XGBRegressor(subsample = 0.8, reg_alpha = 1, random_state = 61, n_estimators = 200, min_child_weight = 1, \
                               max_features = 105, max_depth = 9, learning_rate = 0.1, gamma = 0.1, colsample_bytree = 0.9)
abr_model = AdaBoostRegressor( base_estimator = xgb_model, random_state = 71, n_estimators = 16, loss = 'exponential', learning_rate = 0.1)
```

Voting Regressor

Este es otro de los ensambles que hemos probado, y es nuestro modelo final, pues es el que nos da mejores resultados, pero antes de llegar a la mejor versión pasamos por varias modificaciones y pruebas, las cuales detallaremos a continuación.

A este modelo podemos pasarle la cantidad de estimadores que queramos y adicionalmente los pesos que le damos a cada estimador, luego el modelo evaluará cada estimador y el score final será producto del promedio de los estimadores pero agregándole el factor peso, es decir, dándole mayor importancia a los estimadores cuyos pesos sean más altos.

Lo primero que hicimos fue probar pasándole como estimadores nuestros 2 modelos que daban mejores resultados individualmente, XGBoost y Random Forest, y el resultado fue muy bueno, mejorando el score de ambos modelos por separado.

Competencia de Machine Learning ZonaProp - Organización de Datos

```
xgb_model = xgb.XGBRegressor(subsample = 0.8, reg_alpha = 1, random_state = 61, n_estimators = 200, min_child_weight = 1, \
                               max_features = 105, max_depth = 9, learning_rate = 0.1, gamma = 0.1, colsample_bytree = 0.9)

rf_model = RandomForestRegressor(random_state = 63, n_estimators = 100, min_samples_split = 5, min_samples_leaf = 1, \
                                 max_features = 101, max_depth = 40, oob_score = True)

vr_model = VotingRegressor(estimators = [('XGBoost', xgb_model), ('RandomForest', rf_model)], weights = [1,1])

scores = cross_val_score(vr_model, X, Y, scoring="neg_mean_absolute_error", cv=2)
RF_mae_score = -scores.mean()
RF_mae_score

/usr/local/lib/python3.6/dist-packages/xgboost/core.py:587: FutureWarning: Series.base is deprecated and will be removed in a
  if getattr(data, 'base', None) is not None and \
[18:53:15] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
/usr/local/lib/python3.6/dist-packages/xgboost/core.py:587: FutureWarning: Series.base is deprecated and will be removed in a
  if getattr(data, 'base', None) is not None and \
[19:02:03] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
508096.66160375875
```

Luego de probar este modelo y ver que mejoraba decidimos probar con pasarles otros modelos como estimadores, para complementar a nuestros 2 modelos ya mencionados, los cuales no eran tan buenos pero le asignamos pesos mucho más bajos, y así podríamos evaluar cómo se comportaba el modelo, estos modelos fueron Decision Tree y KNN. Luego de probar diferentes combinaciones para los pesos, mediante Grid Search, y ver que los resultados no eran buenos descartamos la posibilidad de utilizarlos dentro del modelo final.

Finalmente decidimos agregar como tercer estimador un modelo Adaboost, el cual incluye como modelo estimador a XGBoost dentro del mismo, y este es nuestro modelo final.

Luego de determinar cuáles eran los mejores modelos para utilizar dentro de Voting comenzamos a realizar la búsqueda para hallar la mejor combinación de pesos para mejorar el score de nuestro modelo, el resultado final fue el siguiente modelo:

```
xgb_model = xgb.XGBRegressor(subsample = 0.9, reg_alpha = 1, n_estimators = 200, min_child_weight = 1, \
                               max_features = 43, max_depth = 9, learning_rate = 0.25, gamma = 0.0, \
                               colsample_bytree = 0.9)
rf_model = RandomForestRegressor(random_state = 14, n_estimators = 200, min_samples_split = 5, \
                                 min_samples_leaf = 2, max_features = 51, max_depth = 40, oob_score=True)
abr_model = AdaBoostRegressor(base_estimator=xgb_model, random_state = 22, n_estimators = 11, loss = 'linear', learning_rate = 0.4)
vr_model = VotingRegressor(estimators = [('XGBoost', xgb_model), ('RandomForest', rf_model), ('AdaBoo
st', abr_model)], weights = [2,1,3])
```

Este es el modelo definitivo con el cual realizamos las predicciones finales y el que nos llevó al mejor score en Kaggle.

Lasso sobre XGBoost y Random Forest

Este fue el último ensamble probado y consistió en utilizar el regresor lineal Lasso sobre un modelo de XGBoost y otro de Random Forest, para optimizar el tiempo de corrida final y eliminar el AdaBoost que tenía que correr varios modelos de XGBoost. Este regresor le asigna un coeficiente a cada predicción y utiliza un término independiente, con el fin de ajustar los valores finales. Para esto primero hay que entrenarlo con predicciones sobre el set de entrenamiento de cada modelo. Los resultados obtenidos fueron buenos pero no tantos como el modelo del Voting y por eso no se reemplazó ese por este.

Conclusiones

Para dar cierre a este informe decidimos agregar algunas conclusiones sobre cosas que notamos, y que nos parecieron interesantes, a lo largo del desarrollo de este trabajo práctico.

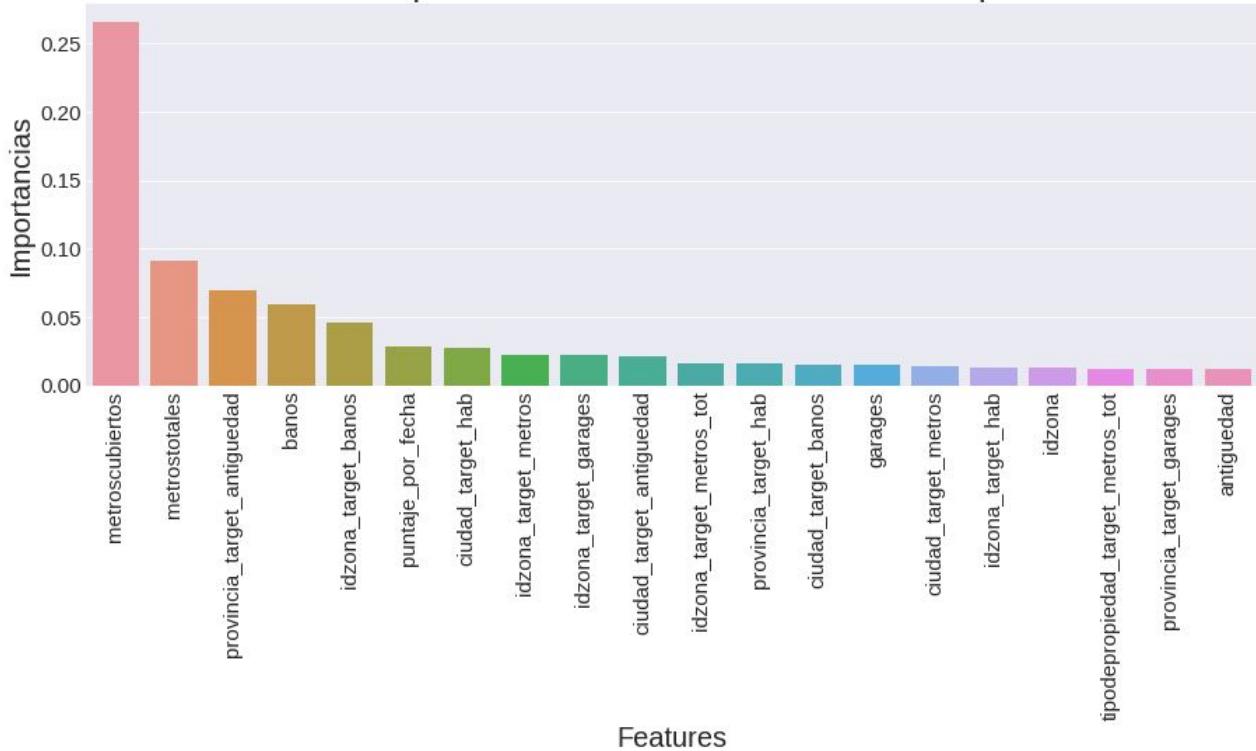
Importancia de Features

Para concluir nos gustaría comparar las importancias de los features utilizados para entrenar nuestros modelos. A continuación compararemos la importancia de los features según dos de los modelos que utilizamos.

Feature Importance Random Forest

A lo largo de todo el Trabajo Práctico fuimos evaluando la importancia de nuestros features de acuerdo a la importancia de Random Forest, de hecho iniciamos viendo la importancia de los features numéricos originales y nos basamos en ello para la creación de nuevos features.

Grado de importancia de los 20 Features más importantes



Notamos que aparecen varios de los features originales y luego también aparecen los features creados con target encoder a partir de ellos, esto resalta de alguna manera la importancia de los mismos, especialmente los metros que ocupan las propiedades, tanto los metros cubiertos como totales tienen un peso muy alto, pues, como ya hemos mencionado en reiteradas ocasiones a lo largo de este informe, están altamente relacionados al precio.

Luego notemos ciertas curiosidades, la antigüedad por sí sola no tiene una importancia demasiado alta, pero al evaluar las antigüedades de las propiedades de acuerdo a las provincias, el peso de la importancia de la misma aumentó mucho, lo mismo ocurre con la cantidad de habitaciones, que de hecho ni siquiera se observa ese feature en el gráfico, pero vemos como la cantidad de

Competencia de Machine Learning ZonaProp - Organización de Datos

habitaciones relacionada con la ciudad, provincia o idzona en la que se encuentra la propiedad son features bastante importantes para este modelo.

Feature Importance XGBoost

Nos pareció interesante agregar además la importancia de los features desde la perspectiva de XGBoost, y tratar de compararlo con la importancia generada por Random Forest.



Vemos que la importancia de los features varía un poco, este modelo le da mucho más importancia al target creado de acuerdo a la antigüedad de las propiedades por provincias, dejando un poco más atrás al feature más importante de Random Forest, metros totales, además vemos como considera features que en el modelo anterior ni siquiera figuraban entre los 20 más importantes. Por lo mencionado en clase sabíamos que la importancia de los features considerada por este modelo no era un reflejo “real” sobre la importancia de los datos en sí, sino que simplemente reflejaba los atributos que el modelo consideraba importantes para generar splits dentro de sus árboles. Esta fue la razón por la cual utilizamos la importancia de Random Forest para contemplar la creación de nuevos features a lo largo del desarrollo de este trabajo práctico. No obstante nos pareció interesante agregar este gráfico para reflejar la diferencia entre los modelos.

Conclusiones Finales

El llenado de nulls en general daba bastantes diferencias en las predicciones, haciendo que mejoren o empeoren, por lo que no es un trabajo trivial. En este set de datos lo que mejor resultado nos dio, logrando una gran mejora, fue llenar los nulls con “lo que es más probable que aparezca”, es decir un promedio en el caso de las columnas con valores numéricos o el valor que más apariciones tuvo en el train para las columnas de variables categóricas.

Los datos de metroscubiertos y metrostotales fueron los más importantes en general para todos los modelos que probamos, sin embargo no siempre son incluidos por los usuarios a la hora de realizar la publicación. Obligar al usuario a incluirlos podría ser una opción para poder predecir mejor los resultados, aunque esto puede que no sea bueno para atraer usuarios, así que otra opción más factible sería tratar de estimar los mismos usando el resto de features, con machine learning o algún otro algoritmo.

Por más que encontramos palabras importantes en descripciones y títulos, para los modelos en general no terminaban siendo muy importantes. Podría hacerse una búsqueda más exhaustiva y detallada de palabras o una mejor opción sería usar algún modelo de redes neuronales para generar clasificaciones de las publicaciones de acuerdo a su título o descripción y esto podría traer mejores resultados aprovechando más estos features.

Tuvimos un estancamiento a la hora de predecir utilizando un solo modelo o un solo ensamble como Random Forest. Luego de probar ensambles entre distintos modelos nos dimos cuenta lo importante que era para lograr mejores predicciones, por las mejoras que producían. También el hecho de buscar hiperparámetros con GridSearch o RandomizedSearch fue muy importante a la hora de mejorar los resultados de las predicciones.

Links a GitHub

Link al Trabajo Práctico Número 2:

<https://github.com/luctiz/Grupo43-TP2>

Link al Trabajo Práctico Número 1:

<https://github.com/luctiz/Grupo43-TP1>