FACULTAD DE INGENIERIA
Universidad de Buenos Aires

75.06/95.58 Organización de Datos

Segundo Cuatrimestre de 2019

Trabajo Práctico II
Machine Learning

Alumno	Padrón	Correo Electrónico
Julián Crespo	100 490	juliancrespo15@gmail.com
Esteban Djeordjian	100 701	edjeordjian@fi.uba.ar
Facundo Fernández	89 843	elfis_@hotmail.com

Fecha de Entrega
5/12/2019

Repositorio del Trabajo
github.com/komod0/TPs-Org.Datos/tree/master/tp2

Contenido

1. Introducción

El planteo dado por la consiga del presente trabajo práctico es el de construir un modelo de machine learning que permita predecir los precios de distintas propiedades, a partir del entrenamiento de un algoritmo con los features adecuados, y teniendo como dataset de entrenamiento las propiedades del trabajo práctico anterior, valiéndose auxiliarmente del análisis exploratorio realizado previamente en el mismo para lograr mejores predicciones.

La propuesta de solución es realizar predicciones a partir de distintos algoritmos, tomando como base aquellos vistos en clase, utilizando la biblioteca de python sklearn, para luego quedarse con el que mejor resultados brinde de forma preliminar (a partir de features iniciales). Por lo tanto, se planteará un preprocesamiento de los datos y feature engineering común a todos los algoritmos, para luego dar un tratamiento especial al dataset para el algoritmo con mejor desempeño a priori, intentando maximizar la calidad de las mejores predicciones alcanzadas (de forma greedy), realizando en paralelo mejoras complementarias a algoritmos secundarios con buen rendimiento.

2. Preprocesamiento de los datos

Lo que se describe a continuación es un conjunto de pasos realizados para la preparación del dataset de cara a ser utilizado por un modelo de machine learning para su entrenamiento. Los mismos son a priori para ser utilizados por todos los algoritmos a probar, por lo que se aclarará en subsecciones correspondientes cualquier variante realizada en el entrenamiento de un algoritmo específico.

La primera determinación es eliminar las columnas del dataset que se entienden como ruidosas para todos los algoritmos. Estas son aquellas asociadas a valores cualitativos e identificadores: *id, dirección, título y descripción*. Para estas últimas dos, se describirán formas de aprovechar su información intrínseca en la próxima sección. Por otro lado, se eliminan las columnas de *latitud* y *longitud* dado su gran cantidad de valores NaN.

Pasando a las columnas con valores numéricos, se quiere rellenar las ausencias de valores para los distintos registros de propiedades. Para el caso de *baños*, *idzona*, *garages* y *habitaciones,* se rellena con la moda por tipo de propiedad y provincia. Por otra parte, los NaN asociados a columnas cuyas distribuciones son asimétricas positivas (positive skewness) se completan con la mediana. Este es el caso para las columnas de *antigüedad*, *metros cubiertos* y *metros totales*. Se evita el uso del promedio por entenderlo como un estadístico sensible a valores extremos, el cual podría tener un peor desempeño para la predicción de los algoritmos.

Para el caso de las columnas categóricas restantes: *tipo de propiedad* y *provincia*, se utiliza la técnica de One Hot Encoding: todos los valores posibles de estos atributos de las propiedades se vuelven columnas, y la presencia de un valor binario para cada registro indica a cuál de todos los casos posibles se encuentra asociada dicha publicación. Ya que se va a trabajar con el nivel de granularidad dado por *provincia* e *idzona,* se omite el uso de *ciudad*.

Finalmente, se separa la fecha de la publicación en *mes* y *año,* omitiendo el valor del día, ya que no se espera que haya una variación del precio con tal granularidad, siendo estos valores enteros. En la figura 2.1 (del informe del trabajo práctico anterior) se puede apreciar que hay una tendencia al aumento del precio promedio, relativamente constante, de forma mensual (junto con algunas caídas y subidas), por lo que probablemente esta diferenciación de los dos valores por separado sea conveniente.

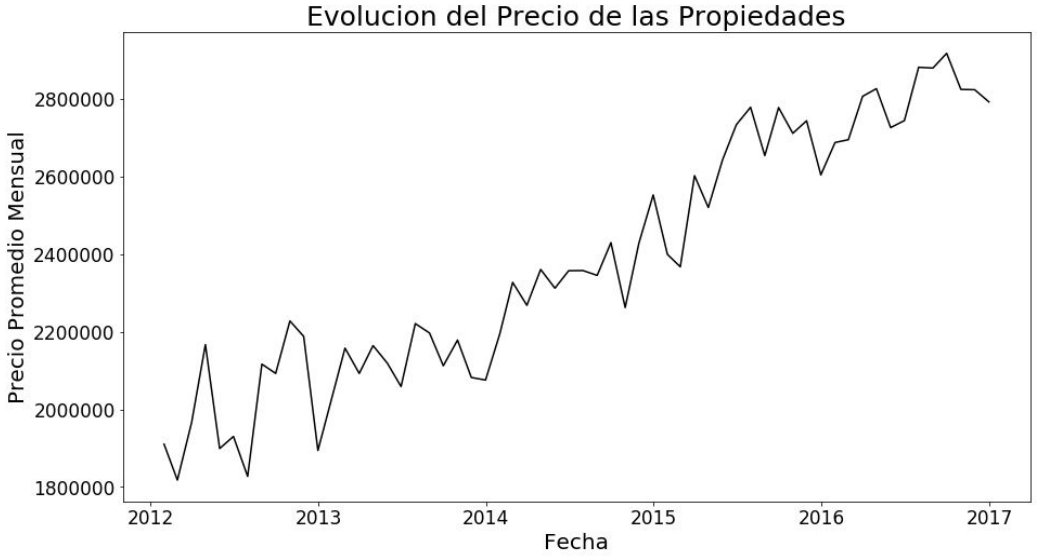

Fig. 2.1.

Se tiene entonces un dataset preprocesado que mantiene todos los registros originales, pero adecuados para el procesamiento de algoritmos de machine learning, según las condiciones establecidas.

3. Feature Engineering

3.1. Features descartados

3.1.1. Dolarización

Se utilizó el sitio web del Banco de México (banxico.org.mx) para obtener la cotización del precio del dólar entre 2012 y 2016, y junto con la fecha de publicación de las propiedades, se hizo el intento de dolarizar los precios al momento de entrenar, para luego re transformar las predicciones a pesos mexicanos. A partir de esta iniciativa no se observaron mejoras considerables. Se maneja la hipótesis de que la conversión sólo consiste en un escalado "especial" de los datos, y que el algoritmo ya puede obtener esta información mediante el precio de la propiedad y la fecha de la publicación de la misma. Dado que realizar este proceso sumaba tiempo de ejecución al pipeline de entrenamiento sin aportar mejoras, se decidió no utilizar los precios dolarizados, y mantener los valores de precios originales.

3.1.2. Cotización del dólar para ese día

Teniendo como referencia la prueba mencionada anteriormente, se hizo un intento adicional de dolarización, pero esta vez agregando una columna con la cotizacion del dolar para cada dia, reflotando el valor de la columna eliminada en el pre procesamiento. Esto tampoco trajo consigo mejoras apreciables (lo cual se entiende a causa del motivo antes descrito), por lo que también se convirtió en un feature abandonado.

3.1.3. Binning de antigüedad

Además de utilizar la antigüedad como feature nativo del dataset original, se intentó aplicarle binning; esto es, se probó agrupar a las antigüedades en buckets de periodos de 5 años, para tratar a la antigüedad como una variable categórica (considerando a cada periodo de dicho lapso como una categoría).

Tomar como "centros de gravedad" a los períodos de 5 años es una idea fundada a partir de lo que se había observado en el trabajo práctico anterior.

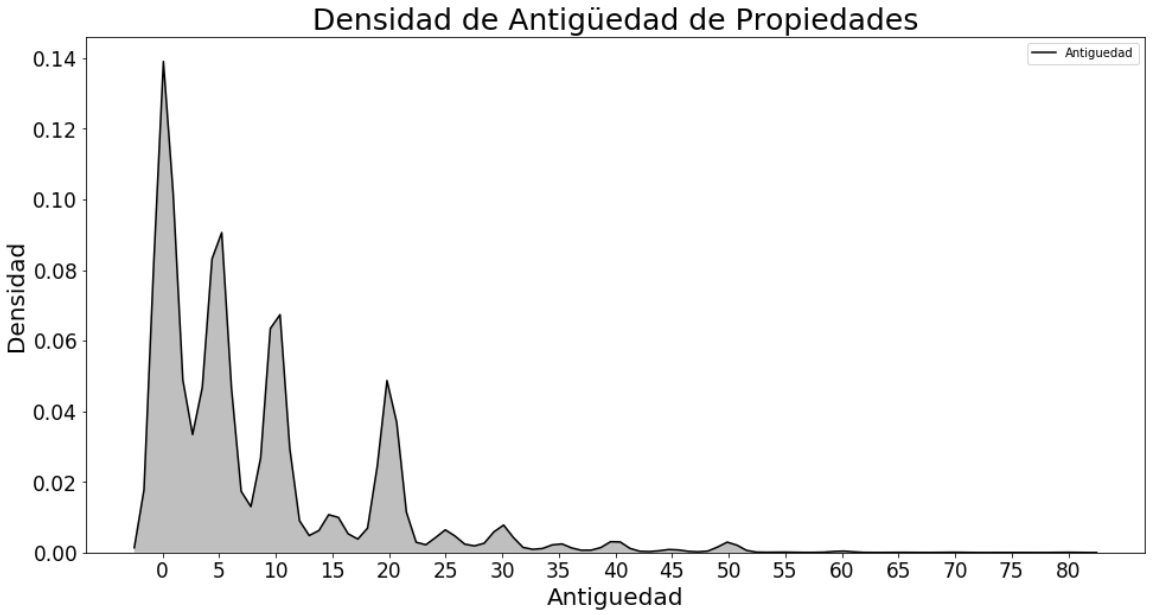

Fig. 3.1.

Esto se hacía bajo la hipótesis de diezmar el posible ruido de las antigüedades intermedias, pero a fin de cuentas no fue un feature que terminara repercutiendo de forma positiva. La hipótesis que se maneja es que haber agregado este feature no sólo disminuyó el nivel de granularidad del feature nativo original, sino que además resultaba conflictivo coexistiendo en paralelo con el mismo, agregando una distorsión que se pretendía evitar.

3.1.4. One Hot Encoding de las ciudades

En principio, se había utilizado la mencionada codificación para todas las ciudades del dataset. En algunas pruebas preliminares, con menos features de los que se describirán a secciones posteriores, se lograron leves mejoras a partir de este aporte. Sin embargo, este feature fue reemplazado por el uso de la población correspondiente a cada ciudad (que se describirá posteriormente), el cual permitió una mejora más apreciable con menos columnas. Por otro lado, se descartó el encoding de ciudades debido a que él mismo parecía eclipsar al feature poblacional en el dataset para la mejora de las predicciones. La hipótesis que se maneja es la de un conflicto similar al descrito en la subsección anterior: el caso de dos columnas que proveían una clase de información similar, pero en un nivel distinto, y en dónde el mejor escenario era dejar sólo la que mayor nivel tenía.

3.1.5. Latent Semantic Analysis sobre las descripciones

Si bien en la sección 2 se comentó cómo *descripción* fue eliminado de por sí como feature por entenderlo ruidoso, se describirá posteriormente cómo parte de su información intrínseca pudo ser reflotada. Entre las formas que se probaron de procesar las descripciones, se menciona a LSA (Latent Semantic Analysis). El proceso de vectorización consistía básicamente en:

- armar una matriz que tenga como filas el TF de cada término presente en el documento correspondiente a esa fila;
- aplicar alguna técnica de reducción de dimensiones a la matriz resultante (en este caso: SVD), siendo dicho número de dimensiones al cual se reduce la matriz variable dependiendo de lo que se resulte manejable en una de las PC con las que se cuenta;
- usar cada una de las columnas de la matriz resultante como feature en el dataset.

Este feature simplemente fue descartado frente a The Hashing Trick, que se describe posteriormente. No

se tiene una explicación formal de porque una de las vectorizaciones haya funcionado mejor que otra.

3.1.6. Número de palabras en las descripciones

Este feature simplemente consistió en indicar el número de palabras que se usaban en la descripción. Se observó que el mismo no repercutió positivamente en las predicciones. La hipótesis que se maneja para explicar esto es que no necesariamente hay una relación entre el precio y la extensión de una descripción, particularmente si se considera el hecho de que varias descripciones tienen dicho campo de texto vacío. Fue más bien un intento económico para sumar información al dataset que no dio buenos resultados.

3.1.7. Cuantiles del precio por cada provincia

Se probaron agregar distintos cuantiles de precio por provincia, a saber: 25, 50, 75. Al igual que con los ya mencionados rechazados, no se percibió ninguna mejora en las predicciones. La hipótesis que se maneja para explicar este caso es que esta información está intrínseca al considerar la provincia de cada registro en los diferentes modelos probados, y agregar una columna específica para esto es sólo una fuente de ruido.

3.2. Features conservados

3.2.1. Transformación del precio

Dado que en el trabajo práctico anterior se había observado que el precio seguía lo que parecía ser una distribución log-normal, se decidió normalizar dicho feature, probando distintas alternativas para esto:

- Transformación del logaritmo: consiste en aplicar logaritmo a la variable para el entrenamiento, para luego deshacer esto en las predicciones (se eleva "e"). Esto se puede lograr (por ejemplo) utilizando numpy.

- Transformación de Box-Cox: una transformación de potencia que utiliza el criterio de la máxima verosimilitud, de forma tal que la distribución resultante se parezca lo más posible a la normal. Se puede encontrar en la librería Scipy.

- Transformación de la raíz cuadrada: se le toma raíz cuadrada a la variable para entrenar, y se deshace dicha transformación en las predicciones (elevando al cuadrado, por ser precios). Algunos algoritmos, como LightGBM, tienen como parámetro la opción de aplicar esta transformación al target antes de entrenar, mientras que para el resto de los casos, se puede realizar de forma manual.

Se observó que transformar los precios logra una mejora sobre las predicciones, probablemente debido al efecto benéfico del escalamiento de una variable para un modelo dado. Por defecto, se utilizó la transformación del logaritmo para el entrenamiento.

3.2.2. Población de las ciudades

Se utilizó la librería Geocoder junto con el proveedor gratuito Geonames para obtener la población de cada ciudad a partir de su nombre. Debido a que los nombres de varias ciudades estaban mal escritos, o estaban escritos en una forma en la que el encoder no las podía interpretar, se utilizó la librería Requests junto con la librería de scrapping BeautifulSoup para buscar en Google el nombre de la ciudad que figuraba en el dataset y obtener la corrección sugerida por Google; todo de manera automatizada. Así, se logró reducir el número de ciudades para las cuales Geocoder no encontraba resultados a solo 8, las cuales fueron rellenadas a mano.

Algo interesante que se observó es que luego de agregar este feature, agregar las ciudades prácticamente ya no tenía impacto sobre las predicciones. Se puede considerar que la población de cada ciudad sirvió entonces como un encoding cardinal de las ciudades. De esta forma, se redujo la cantidad de columnas necesarias para representar las ciudades (con respecto a las aproximadamente 900, si se usaba One Hot Encoding).

3.2.3. Presencia de palabras en atributos descriptivos

Si bien los atributos *título* y *descripción* fueron excluídos de ser features como tales, se analizaron los mismos en búsqueda de palabras claves para generar columnas binarias que sean indicativas de la presencia o no del concepto asociado a dichas palabras. Los términos analizados para *descripción* fueron: *terraza*, *jardín*, *servicio*, *vestidor*, *salon*, *cuarto*, *family*, *hectárea* y *torre*, mientras que para *título* se utilizaron: *bosque*, *avenida*, *fuente* y *golf*.

La selección de estos términos en particular se realizó en base a un análisis de las frecuencias relativas de cada término: se buscó en especial términos que tuviesen una preponderancia mayor en propiedades que se consideraban caras, en contraste con el resto de las propiedades.

3.2.4. THT con atributos descriptivos

Tanto para *título* como para *descripción*, se aplicó The Hashing Trick signado con un número de dimensiones determinado sobre todas las palabras contenidas en cada uno de los mencionados features, dando lugar a un vector indicativo de los resultados de los hash sobre los títulos y las descripciones de cada registro. Esto fue realizado posteriormente de una limpieza de los campos de texto, lo cual incluyó el parseado de los campos (ya que algunas descripciones incluyen tags HTML), remoción de tildes, caracteres especiales, caracteres de fin de línea y similares.

Adicionalmente, se probó utilizar técnicas simples de NLP, entre ellas:

- Stemming: reducir palabras a su raíz, por ejemplo *picante* y *picar* se reducen a su raíz *pic;*
- Keyword extraction: encontrar de forma automatizada las palabras claves o más relevantes en un corpus de textos, para lograr esto utilizamos la librería *rake-nltk;*
- Text summarization: aplicación de textrank para resumir textos, utilizando la librería *summa.*

Con respecto a configuraciones, parámetros y variaciones de The Hashing Trick, se probó:

- funciones de hashing signadas y no signadas;
- Distinta cantidad de dimensiones para los vectores resultantes (300, 600, 700 y 1 000 para el caso de las descripciones, y 50 y 200 para el caso de los títulos);
- Hashear los n-gramas de cada descripción utilizando distintos rangos de longitud para los mismos: por ejemplo se probó hashear todos los n-gramas de caracteres de longitud entre 3 y 4.

Tanto el uso de n-gramas, como los grupos de n palabras, así como las funciones no signadas no parecieron dar resultado, probablemente debido a la menor integridad de los datos con respecto al hashing de las palabras completas, y con una función signada, que ocasionaba un aumento en el número de las colisiones en el vector de salida (al menos, para los tamaños de vectores que se probaron).

Por otro lado, para las pruebas realizadas la cantidad de dimensiones en los cuales se vectorizaban las descripciones pareció ser directamente proporcional a la calidad de las descripciones. Por defecto, para los modelos se utilizó un vector de tamaño 1 000 para el caso de las descripciones, y de tamaño 50 para el caso de los títulos.

3.2.5. Score TF-IDF

Se construyó una query de TF-IDF a partir de palabras que se entienden propias de una propiedad cara, de acuerdo a lo observado en las descripciones de los distintos registros ("jacuzzi", "lujo", "torre", etc.). A continuación, se realizó una matriz TF-IDF con el TfidfVectorizer de sklearn, y se calculó el correspondiente score de cada descripción del dataset contra esta consulta. El valor obtenido de esta forma fue utilizado como feature, bajo la hipótesis de que un score mayor se asocia a propiedades caras.

3.2.6. Features para los modelos de árboles randomizados

Lo que se describe en esta sección son features que fueron exclusivamente utilizados para los modelos de árboles randomizados, en reemplazo del resto de los otros features descritos en esta sección, con las excepciones de la población de las ciudades, y del One Hot Encoding en los tipos de propiedad (aunque esto fue parte del pre procesamiento de datos, como forma de adaptar el dataset para el entrenamiento de los modelos). El motivo de esto es simplemente el acuerdo grupal de tratar a estos modelos con otros features.

Para comenzar, se diverge con el desecho de las coordenadas geográficas, que fueron consideradas como descartadas en la sección de pre procesamiento de datos. Las coordenadas tienen valores nulos en una proporción del 51% para ambos datasets (quizás como parte de un proceso de atenuación de veracidad, para proteger la privacidad de los usuarios de la plataforma), además de datos erróneos. Como ejemplo de estos últimos, en el trabajo práctico anterior se observó que algunas coordenadas se referían a puntos en el Océano Pacífico, muy lejos del territorio de México. Por esto, se taxonomizan estos outliers estableciendo un rango válido para coordenadas de latitud y longitud.

Extremos	Sexagesimal	Decimal	Límite / Frontera
Norte	32° 43′ 06″ latitud norte	32.718333° latitud norte	Estados Unidos
Sur	14° 32′ 27″ latitud norte	14.540833° latitud norte	Guatemala
Este	86° 42′ 36″ longitud oeste	-86.710000° longitud oeste	Isla Mujeres
Oeste	118° 27′ 24″ longitud oeste	-118.456667° longitud oeste	Océano Pacífico

Tabla. 3.1: Coordenadas extremas para el territorio mexicano.

Los valores que cayeron fuera de estos rangos se reemplazaron con np.NaN. De esta forma se pueden eliminar casos que introducen ruido en los datos. Tanto antes de la anulación como después, las coordenadas nulas quedaron en pareja: a los casos que les falta la latitud, también les falta la longitud.

A continuación, se localizan las filas que tienen ciudades nulas y coordenadas válidas. Con un geolocalizador del módulo geopy, se utilizaron esas coordenadas para rellenar las ciudades faltantes. Los casos que no se pudieron completar se rellenaron manualmente. Una vez completadas las ciudades faltantes, se hace lo mismo con las provincias nulas.

A partir de lo anterior, pueden haber quedado ciudades o provincias que refieren al mismo lugar, pero con nombres diferentes. Por ejemplo, en el atributo "provincias" aparecieron los nombres "Edo. de México" y "Estado de México", así como "San luis Potosí" junto con "San luis Potosí". Para el caso de las provincias es fácil de arreglar: sencillamente se hace un reemplazo por los nombres originales. Para las ciudades, se decidió no ahondar en este aspecto, porque son demasiadas como para analizar manualmente.

El siguiente paso es armar una lista con todas las ciudades que aparecen en ambos Datasets y asignarles sus coordenadas, para así reemplazar aquellas ciudades sin latitud ni longitud. Para aquellas filas que no tienen ciudades, pero sí provincias, se usó la moda de ciudad en sus respectivas provincias y luego se usaron las coordenadas de esas ciudades.

Una vez completadas las filas, la cantidad de datos usables en las columnas *provincia*, *ciudad*, *latitud* y *longitud* coinciden. Como complemento, se puede rellenar los primeros dos features anteriores con la provincia moda y la ciudad moda.

Adicionalmente, siendo las ciudades y las provincias variables no numéricas, se deben codificar de alguna manera para entrenar a los modelos. Pero las ciudades no fueron incluídas en el pre procesamiento de datos antes descrito, siendo que superan la cantidad de 900. En consecuencia, se decide incluirlas utilizando Binary Encoder (del módulo "category_encoders"), ya que usar One Hot Encoding implicaría un consumo excesivo de recursos de memoria.

Se rellenaron los nulos de *tipo de propiedad*, también con modas según ciudad, y se codificó esta feature utilizando One Hot Encoding. Por otra parte, los faltantes de *idzona* se completaron por moda de ciudades primero y moda de provincia luego.

Finalmente, los otros dos features utilizados son el uso de una columna que represente la suma de los metros totales y cubiertos, y una columna para cada uno de los features que indica qué casos fueron originalmente nulos. Para este último, básicamente se confeccionaron columnas binarias que tienen 1 en aquellas filas que registran datos nulos en su forma inicial. Esto se realizó para todas las columnas que presentaron este tipo de obstáculo. Con estas nuevas feautures se consiguió una leve, pero consistente, mejora en la predicción en varias pruebas.

4. Algoritmos Probados

4.1. Linear Regression

Este algoritmo simplemente intenta ponderar el peso de cada uno de los atributos del set de entrenamiento en una combinación lineal, y su objetivo es aprender los parámetros asociados a dichos pesos. No se espera que un modelo tan simple tenga buenos resultados de las predicciones, y se lo utiliza más bien como un algoritmo preliminar para iniciar el trabajo. En particular, se utiliza para este modelo una heurística de valores extremos: todos los valores inferiores al menor valor conocido de una propiedad ($ 310 000, en pesos mexicanos) o superiores al mayor precio visualizado ($ 12 500 000) son convertidos a las cotas respectivas. Esto último evita predicciones absurdas por parte de este modelo (como precios negativos), y outliers (precios excesivos).

Los mejores resultados obtenidos a partir de este modelo se lograron utilizando todos los features descritos en la sección anterior, y en términos de una validación contra el propio set de entrenamiento, con un 33% del set dedicado al test, se obtuvo un error promedio de aproximadamente 806 000.

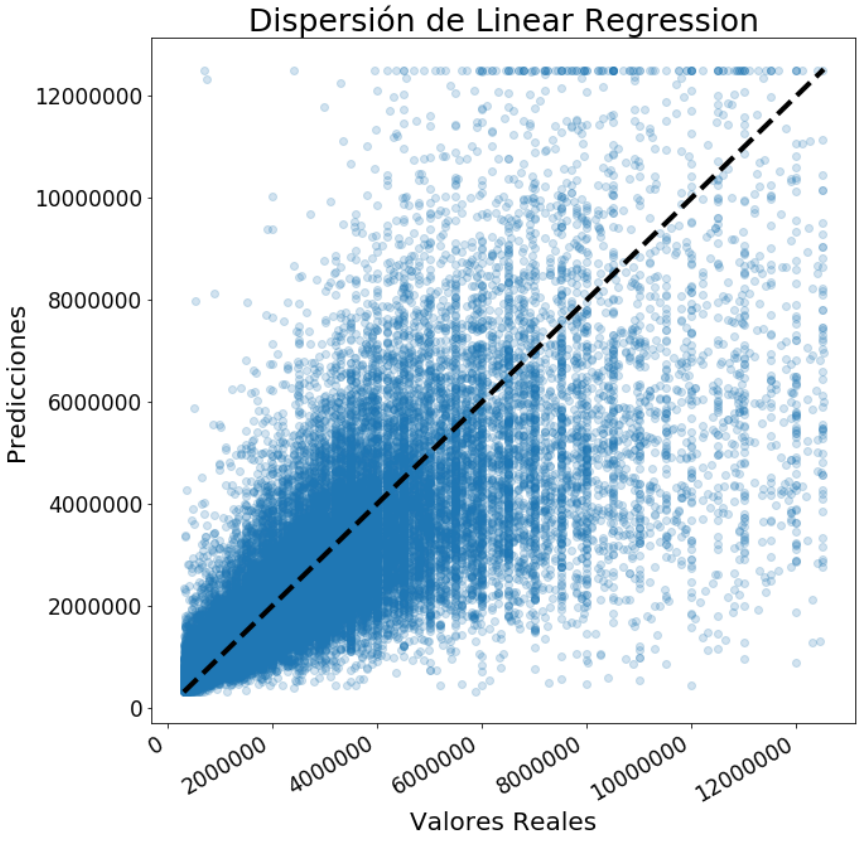

Fig. 4.1.

Esta propuesta de utilizar una fracción del dataset de train para validar los resultados con la fracción restante en contraposición a un K-Fold Cross Validation más ortodoxo se justifica debido a que el mencionado proceso ralentiza mucho el proceso de prueba de features y validación de modelos (incluso para valores bajos de K, como K = 5), y en contraposición, el uso de una fracción del train utilizando la métrica indicada en la consiga (error absoluto promedio, o MAE) brinda resultados de error muy representativos con respecto a realizar una subida a Kaggle. En consecuencia, se prefiere y mantiene esta forma de verificación para el resto de los modelos como opción por defecto.

Como comentario adicional, se menciona que la normalización de los datos en el uso del algoritmo no parece haber tenido impacto significativo en el resultado de las predicciones.

4.2. KNN

Para el algoritmo de búsqueda de vecinos más cercanos, se probaron las distancias: Euclídea, Manhattan, Minkowski (con hiper parámetro "p" distinto de 1 y 2), Coseno, Jaccard y Hamming. Todas aquellas no asociadas a Minkowski resultaron asombrosamente lentas, a pesar de estar paralelizadas en 12 hilos, y no lograron en ningún caso resultados mejores que la Manhattan, la cual fue la que menos error promedio arrojó: 709 000.

En particular, se observó que las distancia de Jaccard se comportó particularmente mal como métrica de predicciones, mientras que la de Minkowski dio las mejores predicciones para este modelo (y de forma bastante rápida), mejorando estas a medida que se disminuyó el valor de "p" (obteniéndose para su valor 1, el mejor resultado posible). Otra observación de interés podría ser que el error en las predicciones parecía disminuir con el aumento de la cantidad de vecinos. Este fenómeno se observó hasta el valor aproximado de 40 vecinos (el cual fue utilizado finalmente), valor a partir del cual había un estancamiento en la mejora para todas las distancias, y luego del cual volvía a haber un empeoramiento de las predicciones (a partir de valores mayores a 200 vecinos, por ejemplo). Como notas adicionales, el "tamaño de hoja" no pareció repercutir en los resultados, y se observó lo comentado en clase: en este modelo lo rápido es "entrenar", no predecir (ya que justamente el algoritmo lo que realiza es la predicción de los vecinos similares).

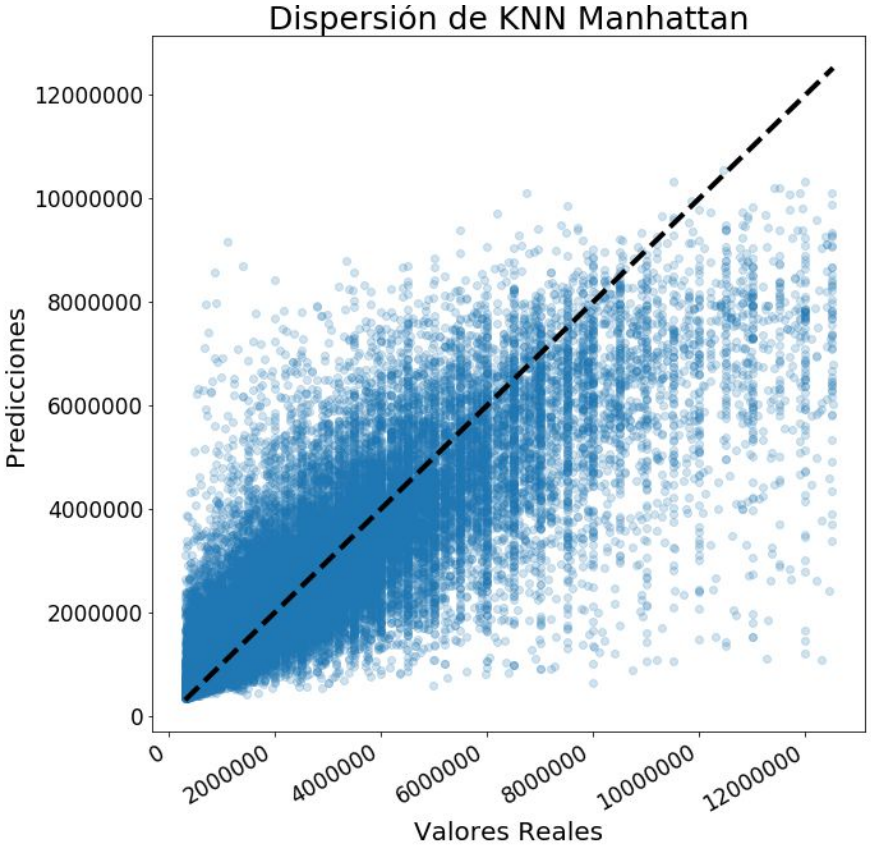

Fig. 4.2.

4.3. Perceptron

Se intenta en este caso utilizar la versión de regresión del Perceptron multi capa de sklearn: MLPRegressor, normalizando los datos con Standard Scaler. El tunning fino sobre este modelo fue desalentado por la conocida observación empírica de que un modelo de redes neuronales difícilmente pueda tener mejor rendimiento que un algoritmo de boosting basado en árboles.

En última instancia, utilizando la función de activación logística, con 10 000 iteraciones y un learning rate inicial de 0.01 (es decir, la conocida combinación final de bajo learning rate y un alto nivel de iteraciones), se logró tener un error absoluto promedio de algo menos de 770 000. Esto es claramente un mal resultado para el algoritmo, y existe un consenso generalizado en el grupo que este valor podría haberse mejorado mucho (de hecho, se tuvo que usar la heurística de los extremos de Linear Regression), pero el hecho es que ese tiempo y bastante más fue dedicado a un tunning más fino sobre los modelos de boosting, que desde el primer momento respondieron mejor que este y los anteriores modelos descritos.

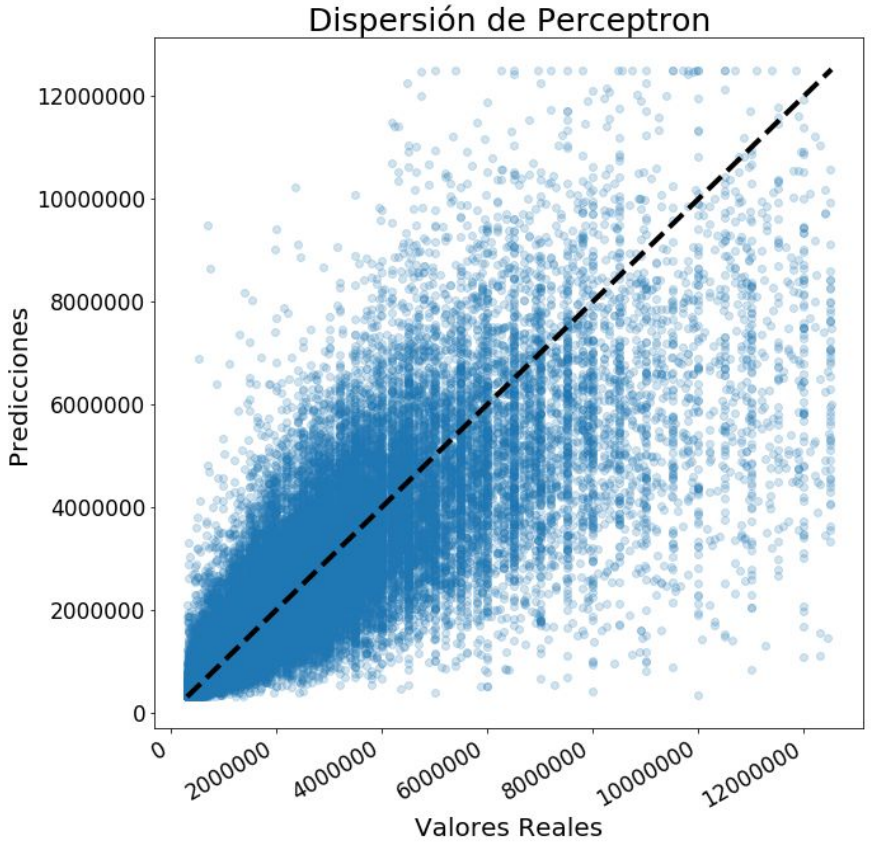

Fig. 4.3.

4.4. SVM

Si bien SVM es bien conocido por ser un algoritmo de clasificación (siendo su implementación en sklearn SVC), también cuenta con una versión para problemas de regresión (cuya implementación en sklearn es SVR), que fue probada para las predicciones de precios, aunque la elevada cantidad de tiempo que se debe esperar para obtener resultados con este algoritmo (incluso en la predicción), sumado a los malos resultados intermedios, aún con los datos escalados, desalentaron bastante las diferentes pruebas con distintas columnas.

Lo primero que se nota con este algoritmo es que (al menos en la implementación de sklearn) es realmente lento, y no es paralelizable. Según las pruebas realizadas, de los diferentes hiper parámetros disponibles para la configuración, el que afectaba notablemente a la velocidad de ejecución era la cantidadde iteraciones. Por defecto, no existe un límite para las mismas, y esto parece ser lo que causa el

elevado nivel de tiempo que se debe esperar para obtener el resultado de una ejecución. Por supuesto, establecer una cantidad de iteraciones baja (100) resulta en predicciones casi inmediatas pero muy imprecisas, mediante que una cantidad de iteraciones alta (100 000) no permite visualizar resultados sino en varias horas. El equilibrio encontrado para el desempeño de este algoritmo es de 10 000 iteraciones, obteniendo aún así el resultado de un error absoluto promedio aproximado de 840 000.

Por otra parte, de los tres "algoritmos" (kernels) para utilizar con SVM: polinomial, radial y sigmoide, se encontraron los "mejores" resultados para radial. Una observación que se puede destacar sobre la elección en el grado del polinomio, durante el marco de las pruebas, es su errática proporcionalidad con la calidad de las predicciones (subirlo solía mejorar los resultados, pero esto no siempre se cumplía, y grados bastantante diferentes producían resultados similares).

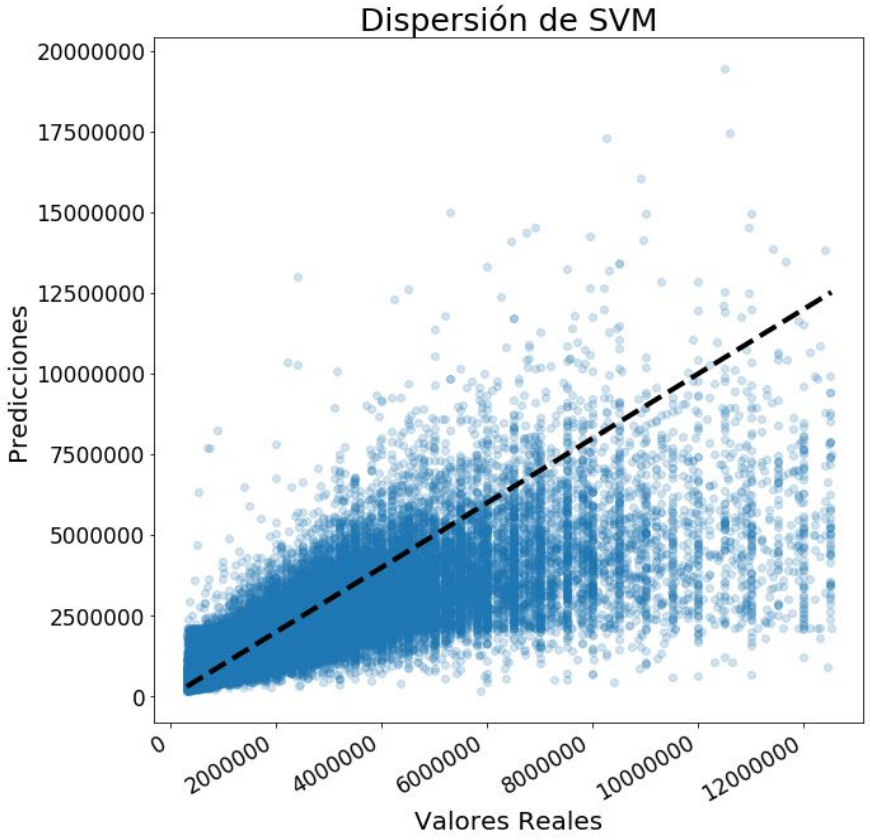

Fig. 4.4.

4.5. Gradient Boosting

Los intentos de predicciones continúan con el modelo más simple de booster. La naturaleza del algoritmo está alineada con los problemas de regresión, por lo que se entiende que puede dar buenos resultados para el problema de la predicción de precios. Sin embargo, este modelo utiliza una función de pérdida como forma de aproximación para la minimización del error, y no es paralelizable, por lo que se lo entiende sólo como un escalón de referencia para llegar a modelos de boosting algo más avanzados. En este sentido, el tunning del algoritmo no fue profundo, ni detallado, siendo particularmente desalentado por el enorme tiempo de espera para lograr un resultado. De hecho, fue tal el tiempo de espera que los features descritos en la sección correspondiente no pudieron ser utilizados, lo cual derivó en una retrotracción del modelo a los features utilizables exclusivamente a partir del pre procesamiento.

Usando la cantidad por defecto de estimadores, si no se impone una restricción sobre la condición de corte sobre el modelo, el mismo puede estar varias horas sin dar resultado alguno. Incluso bajo los limitantes mencionados (que específicamente fueron utilizar "n_iter_no_change = 5" y "tol = 0.1"), y solo utilizando un número muy bajo de estimadores (5) se pudieron obtener resultados preliminares en tiempos razonables, los cuales terminaron por ser definitivos debido a todos los obstáculos para obtener un simple

vista preliminar del potencial del modelo. El error promedio para la mencionada ejecución (bajo el ya mencionado método del sample sobre el archivo de train) fue el paupérrimo valor de 1.2 millones (luego de más de dos horas de ejecución). Esto se entiende como un severo caso de underfitting, comprobando que el modelo no es adecuado para el problema.

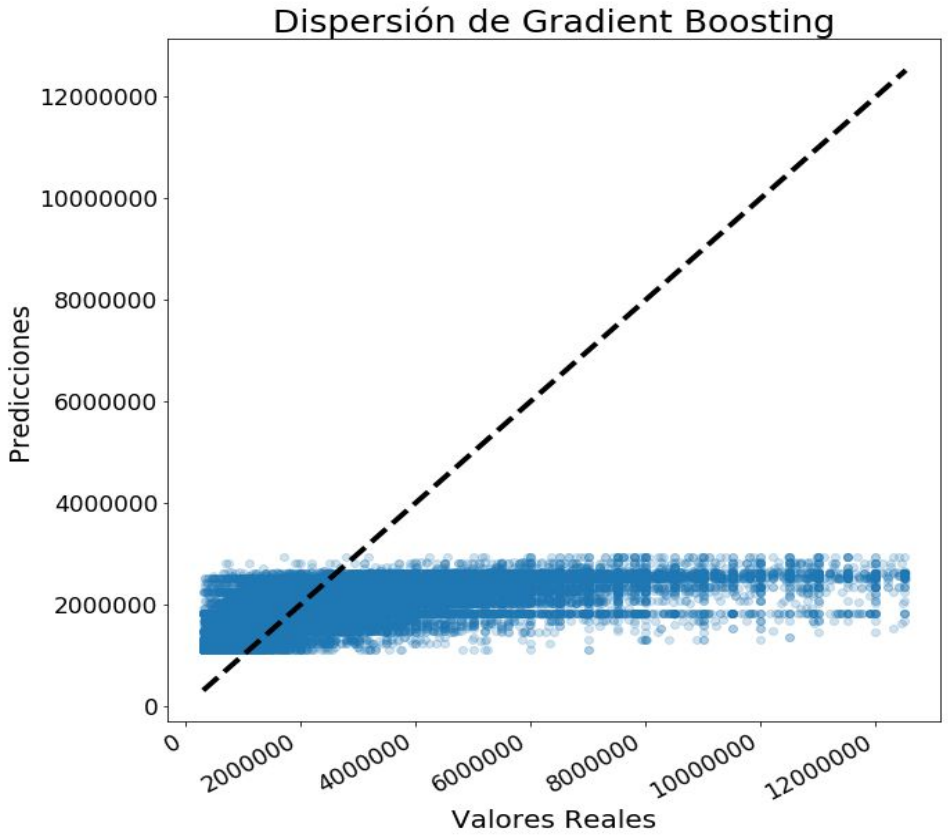

Fig. 4.5.

Hasta aquí, se han presentado los modelos que no fueron exitosos para las predicciones. Esto en parte puede que sea porque no son los algoritmos apropiados para el caso (como podría ser la situación para LinearRegression y SVD), o bien, porque si bien son modelos con mucho potencial (como Perceptron), los malos resultados obtenidos de forma preliminar en contraste a los algoritmos siguientes a comentar, aún en etapas muy preliminares del desarrollo de features, esfumaron completamente la posibilidad de concentrar esfuerzos en estos modelos, y en cambio, se puso el foco en los modelos a describir a continuación (de acuerdo a la estrategia greedy comentada en la introducción).

4.6. Modelos de árboles aleatorios

4.6.1 Random Forest

Se utilizó el modelo de Árboles Aleatorios (Random Forest) RandomForestRegressor del módulo ensemble. Los mismos se arman a partir de subconjuntos de los features totales, lo cual evita el overfitting. El algoritmo es paralelizable, por lo que los tiempos de entrenamiento son adecuados para realizar distintos tipos de pruebas.

Los hiper parámetros básicos de este modelo, número de features y árboles a usar, parece a priori directamente proporcionales a la calidad de la predicción. Sin embargo, esa mejora sufre un amesetamientopara valores suficientemente elevados que lleven a un gran consumo de recursos (memoria y tiempo), lo cual se asemeja a lo que ocurre con otros modelos para con la cantidad de iteraciones (como LGBM).

Para evaluar este modelo en particular, se usó un K-Fold de K=10. El mismo divide el set de datos en 10 partes iguales, toma una como target, y entrena con el resto del set. Se van tomando sucesivamente las distintas partes de target, y se entrena con la restante, teniendo 10 entrenamientos en total. Para cada etapa se registró el error promedio absoluto y el score en vectores para cada cantidad. Al final, se estimó un error promedio y un score promedio de entre todos esos valores.

Adicionalmente, se realizó una selección de features especial para comparar el tiempo de cálculo y la diferencia entre predicciones con sets de datos completos, y sets con features elegidas según su importancia. Se realizaron dos análisis para observar la variación en la exactitud en la predicción: en uno se usó una variación en el número de estimadores, y en el otro una variación en la cantidad features.

	MAE		Score		
# Árboles	Promedio [$]	Desvío std [$]	Promedio	Desvío std	Tiempo
5	614873	4671	0.758	0.004	0 min 52 seg
	616576	3317	0.758	0.003	0 min 40 seg
10	582412	5203	0.782	0.003	1 min 41 seg
	583265	4249	0.781	0.003	1 min 13 seg
20	563645	3339	0.795	0.002	3 min 11 seg
	565624	4302	0.794	0.003	2 min 08 seg
40	554912	3531	0.800	0.003	5 min 38 seg
	556249	3974	0.800	0.003	4 min 09 seg
60	551432	4289	0.802	0.003	8 min 49 seg
	552807	4202	0.802	0.003	5 min 50 seg
80	550215	4463	0.803	0.003	11 min 22 seg
	551089	4106	0.803	0.003	8 min 11 seg
100	549599	4292	0.803	0.003	14 min 00 seg
	550342	4172	0.803	0.003	9 min 52 seg

Tabla 4.1: mediciones de Random Forest para distintas cantidades de estimadores.

Para MAE y Score, se muestran los promedios y desvíos estándar obtenidos de las predicciones realizadas en 10 folds. Las predicciones se realizaron con 96 features de entrenamiento y el precio. En fondo oscuro, se muestra el resultado de una selección de features que superan un mínimo de 0.001 de importancia en una corrida con 100 árboles: se encontraron 45 features importantes (ver fig. 4.5).

		MAE		Score		
Caso	# Features	Promedio [$]	Desvío std [$]	Promedio	Desvío std	Tiempo
1	11	930698	5189	0.542	0.004	2 min 44 seg
	11	930698	5189	0.542	0.004	2 min 44 seg
2	35	830863	5666	0.615	0.004	4 min 46 seg
	21	831781	5596	0.615	0.005	3 min 42 seg
3	78	636822	5069	0.747	0.003	10 min 53 seg
	39	641156	5299	0.745	0.003	6 min 38 seg
4	80	598198	5063	0.772	0.004	11 min 45 seg
	36	599537	5056	0.772	0.003	6 min 54 seg
5	92	583082	5387	0.784	0.004	13 min 28 seg
	43	584963	5168	0.784	0.004	8 min 35 seg
6	94	578993	4707	0.787	0.003	13 min 48 seg
	42	580477	4451	0.786	0.003	8 min 45 seg
7	96	549599	4292	0.803	0.003	14 min 00 seg
	45	550342	4172	0.803	0.003	9 min 52 seg
	76	549282	3954	0.804	0.003	12 min 50 seg

Tabla 4.2: mediciones de Random Forest para distintas cantidades de features.

En fondo oscuro se muestra el resultado de una selección de features que superan un mínimo de 0.001 de importancia (depende de la cantidad de features usada en la corrida). En la última fila se realizó una selección con una tolerancia de hasta 0.0001 de importancia para 96 features.

Los números de los casos anteriores corresponde a la consideración (incremental) de los siguientes features:

1. *gimnasio*, *usosmultiples*, *piscina*, *escuelascercanas*, *centroscomercialescercanos*, *antigüedad*, *habitaciones*, *garages*, *baños*, *metroscubiertos* y *metrostotales*;
2. *tipodepropiedad*;
3. *provincia* y *ciudad* (en los formatos de encoding mencionados);
4. *idzona* y población de cada ciudad;
5. columna indicadora de features inicialmente nulo;
6. *latitud* y *longitud*;
7. *mes* y *año*.

En el siguiente gráfico se puede observar la importancia de los features que superaron el umbral de importancia de 0.001:

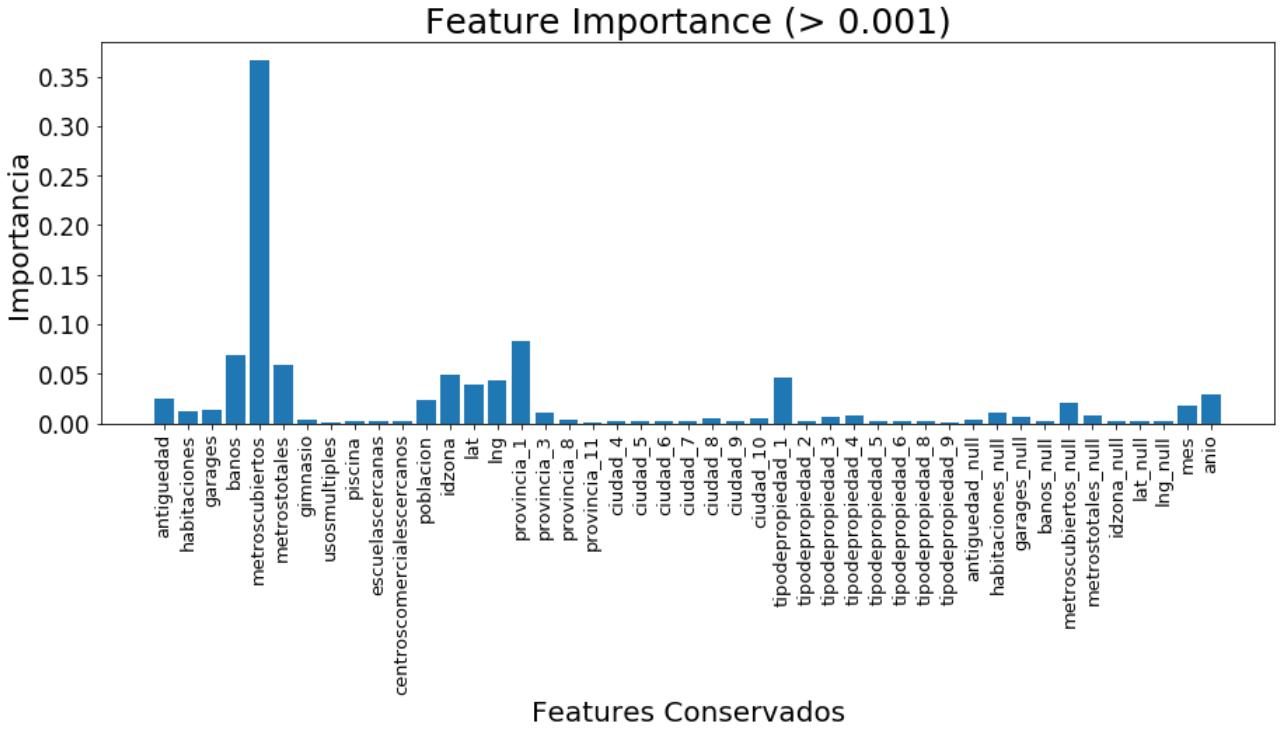

Fig. 4.6: highlights del análisis de importancia de features para Random Forest.

La fuerte presencia de los metros cubiertos dentro de los features importantes no solo es intuitiva, sino que se vincula con la relación lineal que este atributo de las propiedades tenía (aproximadamente) con el precio de las mismas. En particular, es el único feature que coincide en importancia con una prueba de feature importance realizada manualmente (no incluída en el informe por su heterodoxia) con Linear Regression, en la cuál se veía que, de todos los atributos, metros cubiertos era aquel que más empeoraba el valor del error al ser excluído (por sí solo, en contraste con el resto).

Se tiene entonces un modelo que, habiendo sido entrenado con su propio conjunto de features, obtiene el nada despreciable valor de error de 550 000.

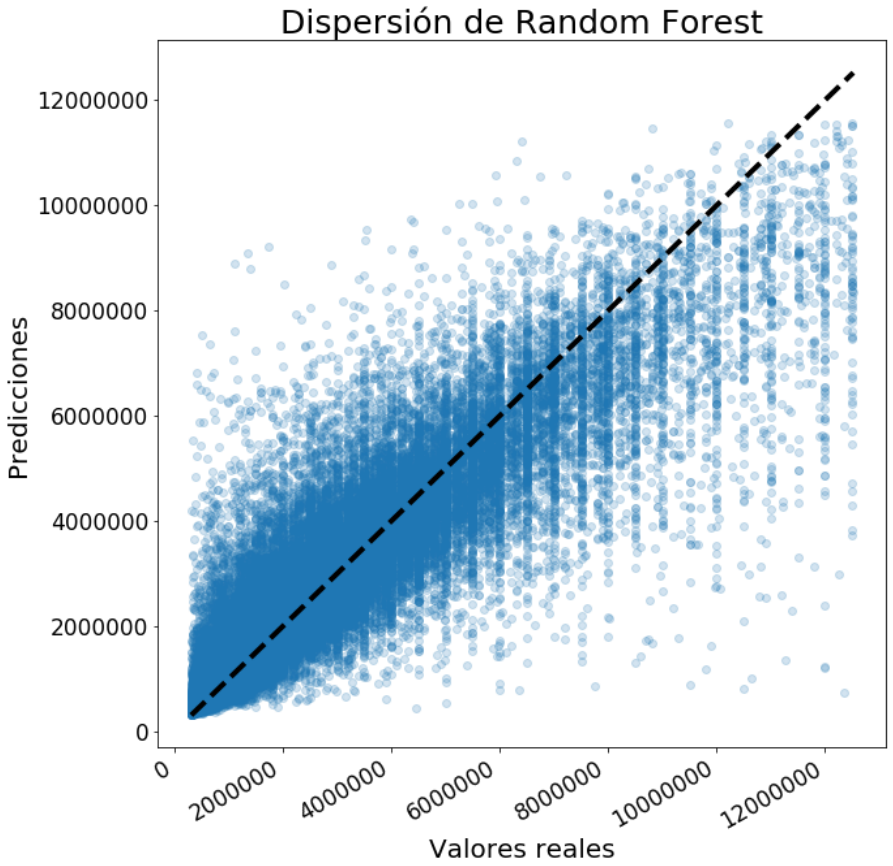

Fig. 4.7.

4.6.2. Extremely Randomized Trees

Se utilizó el modelo de Árboles Extremadamente Aleatorios ExtraTreesRegressor, del módulo ensemble. La diferencia con Random Forest es que cuando elige un feature y un valor que determina el corte a realizar (threshold), selecciona un valor al azar en lugar de elegir con un método que aporte una mayor ganancia de información.

Se utilizó la misma metodología de análisis de Random Forest, y los resultados se muestran a continuación.

	MAE		Score		
# Árboles	Promedio [$]	Desvío std [$]	Promedio	Desvío std	Tiempo
5	612772	4622	0.757	0.003	0 min 53 seg
	617388	5852	0.755	0.005	0 min 46 seg
10	581511	4379	0.780	0.003	2 min 21 seg
	585138	4443	0.779	0.004	1 min 21 seg
20	566138	5670	0.791	0.004	3 min 52 seg
	570113	4792	0.790	0.004	2 min 26 seg
40	557689	4431	0.796	0.003	8 min 14 seg
	560300	4642	0.795	0.004	5 min 06 seg
60	554721	4830	0.798	0.003	12 min 41 seg
	556959	4718	0.798	0.004	6 min 42 seg
80	553192	4563	0.799	0.004	15 min 35 seg
	556191	4941	0.798	0.004	9 min 23 seg
100	552599	4364	0.800	0.003	19 min 09 seg
	555452	4344	0.799	0.003	11 min 31 seg

Tabla 4.3: mediciones de Extremely Randomized Trees para distintas cantidades de estimadores (árboles).

En fondo oscuro se muestra el resultado de una selección de features que superan un mínimo de 0.001 de importancia en una corrida con 100 árboles: se encontraron 46 features importantes. La interesante coincidencia en el número de features probablemente se deba tanto a la similitud del análisis, como a la similaridad entre los modelos de árboles randomizados.

		MAE		Score		
Caso	# Features	Promedio [$]	Desvío std [$]	Promedio	Desvío std	Tiempo
1	11	968312	6219	0.491	0.007	2 min 26 seg
	11	968312	6219	0.491	0.007	2 min 26 seg
2	35	862316	5930	0.576	0.005	5 min 15 seg
	21	863171	5482	0.575	0.006	3 min 42 seg
3	78	640078	5199	0.736	0.004	13 min 56 seg
	41	642940	4971	0.735	0.003	8 min 21 seg
4	80	615210	4832	0.755	0.004	15 min 29 seg
	39	618170	5115	0.754	0.004	8 min 09 seg
5	92	598200	6215	0.770	0.005	17 min 26 seg
	46	601354	5990	0.769	0.005	9 min 52 seg
6	94	590019	5660	0.776	0.004	18 min 24 seg
	48	592200	6044	0.775	0.005	10 min 41 seg
7	96	552599	4364	0.800	0.003	19 min 09 seg
	46	555452	4344	0.799	0.003	11 min 31 seg
	87	552546	4737	0.799	0.003	18 min 22 seg

Tabla 4.4: mediciones de Extremely Randomized Trees para distintas cantidades de features.

Nuevamente, en fondo oscuro se muestra el resultado de una selección de features que superan un mínimo de 0.001 de importancia (depende de la cantidad de features usada en la corrida). En la última fila se realizó una selección con una tolerancia de hasta 0.0001 de importancia para 96 features.

En el siguiente gráfico, análogo al modelo anterior, se puede observar la importancia de los features que superaron el umbral de importancia de 0.001:

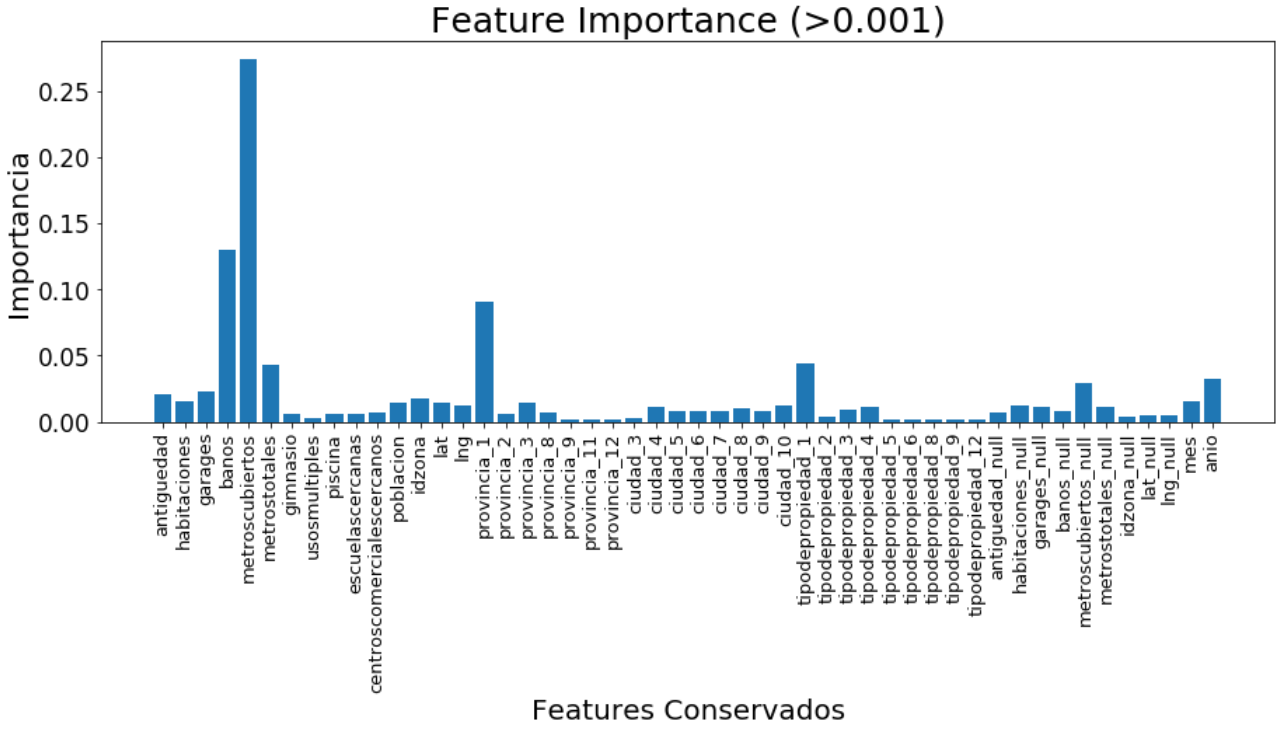

Fig. 4.8.

Este modelo arroja un error similar a Random Forest: 560 000.

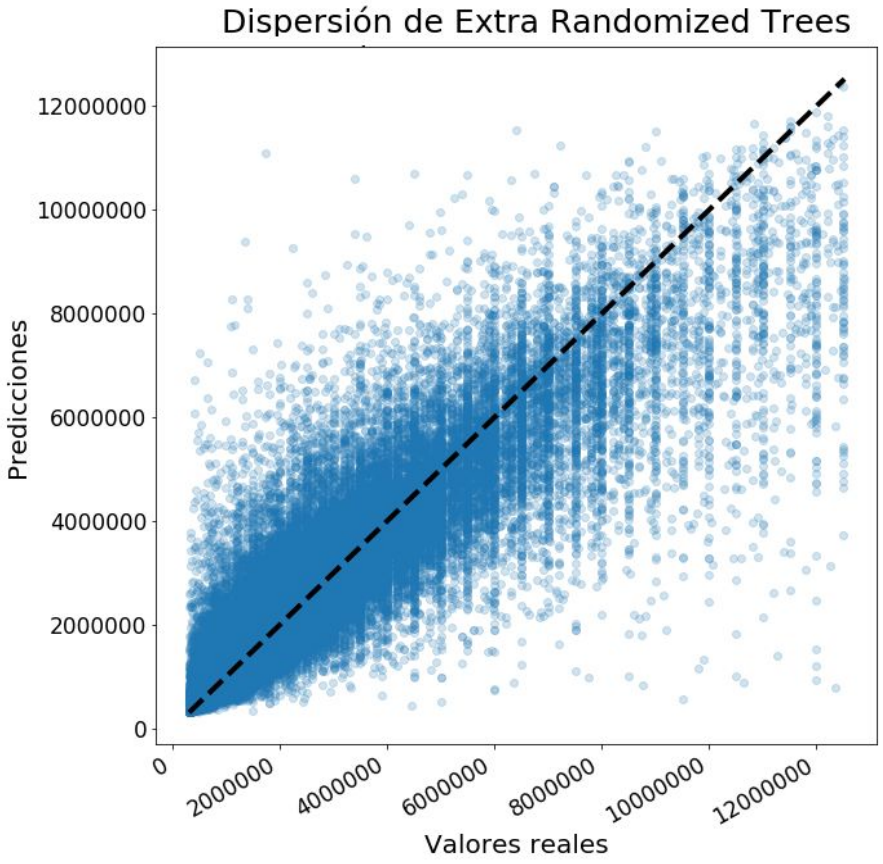

Fig. 4.9.

4.6.3. Comparación entre métodos

De las tablas de mediciones de predicciones obtenidas con Random Forest y Extra Randomized Trees, se pueden construir una comparación entre ambos métodos.

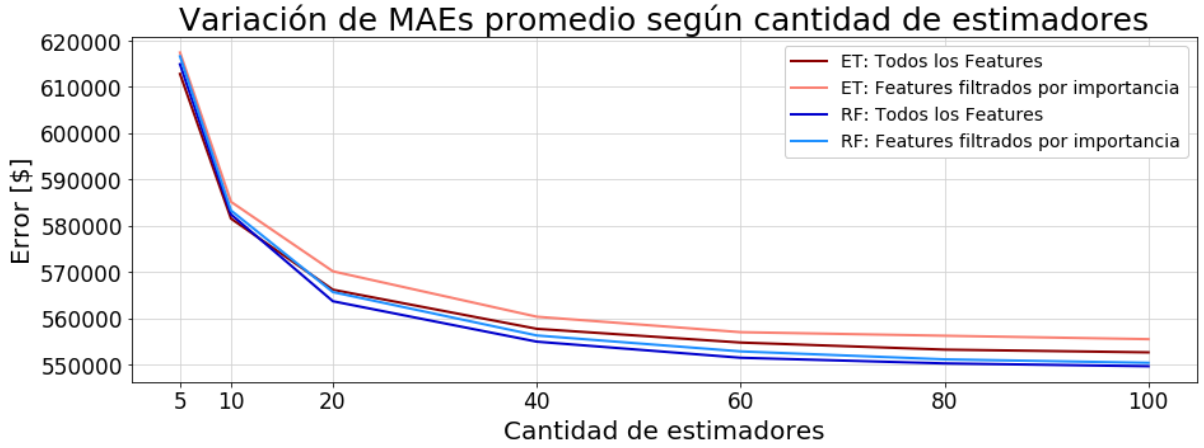

Fig. 4.10.

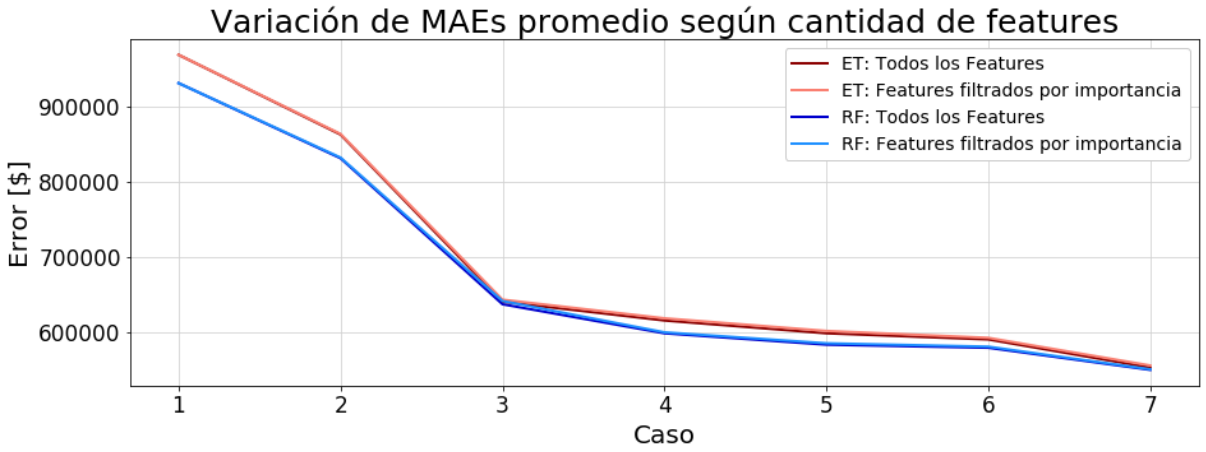

Fig. 4.11.

Se ve que el aumento de árboles y de features mejoran las predicción, siendo Random Forest el modelo que arroja mejores predicciones en ambos escenarios. La cola de la derecha muestra una meseta, donde se destaca que el consumo de recursos comienza a ser considerable. También se observa que las líneas de los métodos que usan los features filtrados tienen un poco más de error que los métodos que utilizan todas las columnas (en la Fig. 4.11 las trazas están más juntas porque se calculó la importancia de features para cada caso, mientras que en la fig. 4.10 las columnas importantes se mantuvieron constantes).

En la figura 4.11 se puede apreciar que los distintos features tienen diferentes niveles de importancia; los ángulos de los "codos" no tienen monotonía: se abren y se cierran.

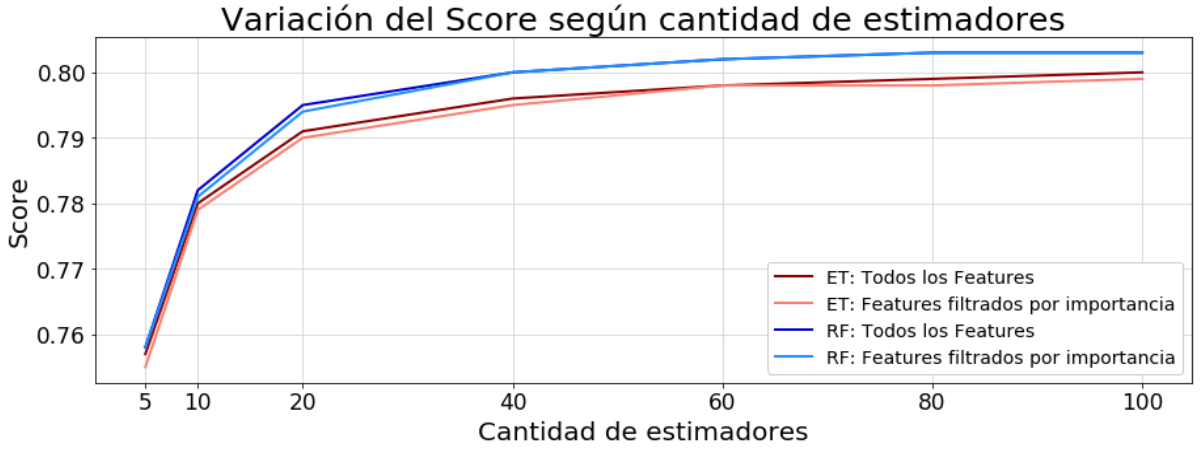

Fig. 4.12.

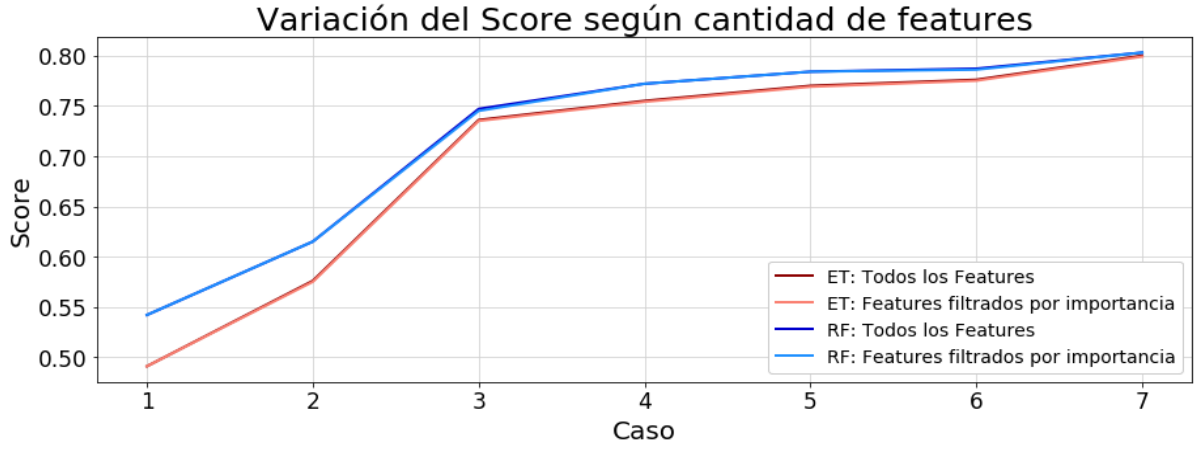

Fig. 4.13.

Por lo visto en los gráficos de errores, era esperable que el score mejore con el aumento de árboles y de features. Nuevamente se ve que el cálculo de importancia de features para cada caso provoca una gran similitud entre los scores de los métodos que usan todas los features, y aquellos para los que fueron filtradas.

Una observación inmediata es que el error y el Score están estrechamente relacionados: cuando el error es alto el score es bajo, y viceversa (uno es inversamente proporcional al otro).

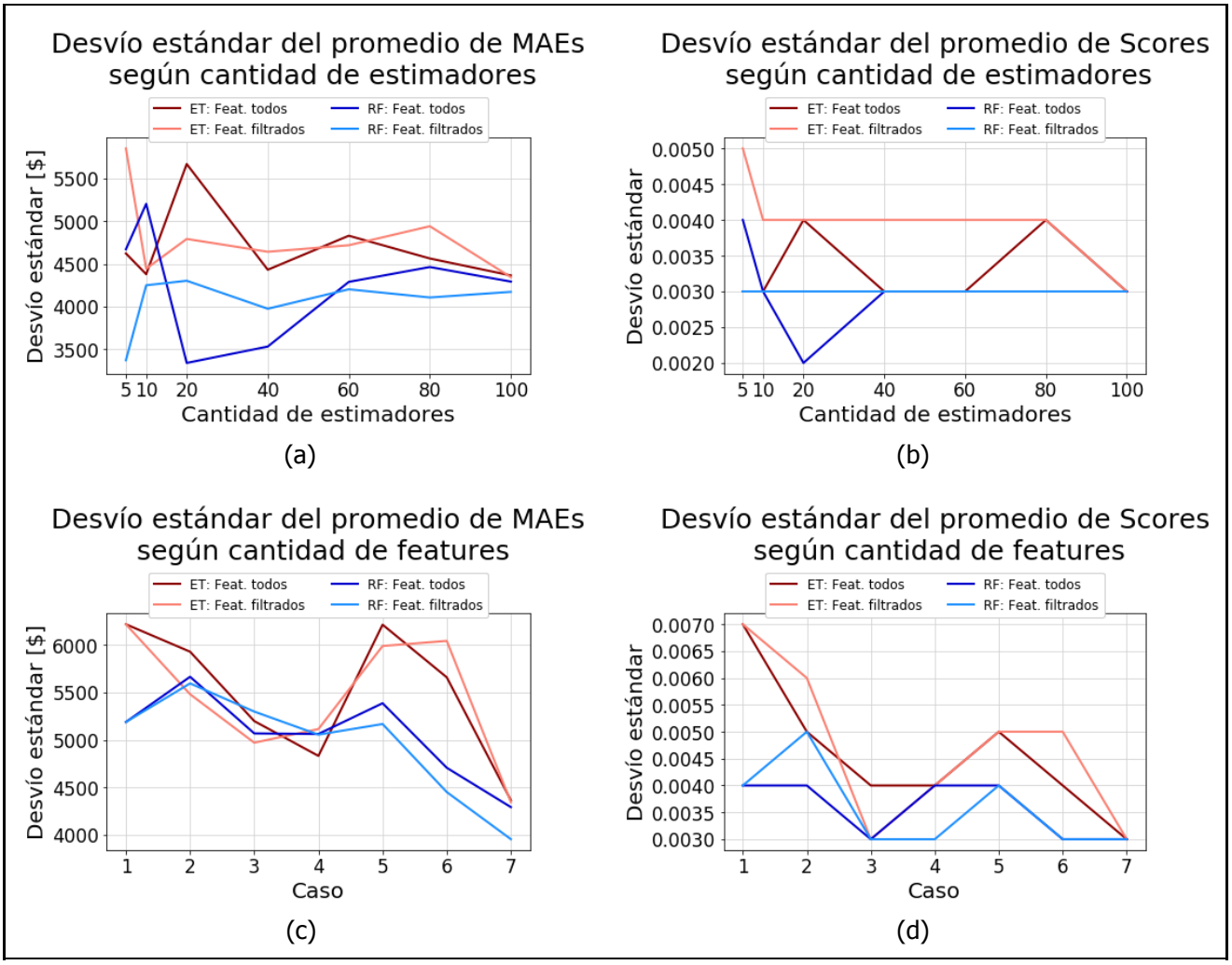

Fig. 4.14: desvíos estándar del MAE y el Score para variaciones de árboles y de features.

En el gráfico (a) los desvíos se estabilizan a mayor cantidad de árboles para los dos algoritmos con todos los features, o bien, solo los seleccionados. Algo similar ocurre en (b). Entonces se deduce que al utilizar la misma información para cada modelo, si se cambia la cantidad de estimadores, la variación del error se estabiliza a medida que éstos aumentan.

Para el aumento de features, que se pueden visualizar en los gráficos (c) y (d), el desvío de los errores y de los Scores no solo se estabiliza, también disminuye. De ésto se puede inferir que la variación en el error disminuye cuando se agrega más información al modelo (así como también lo hace su desvío estándar). En (c) se puede ver que en el caso 5 el desvío aumentó al agregar la información de los datos nulos. En la figura 4.11 se nota una leve disminución en el error (caso 5 en tabla 4.4), pero se introduce algo de ruido con los nuevos datos.

Si se observan los gráficos de los errores, en los gráficos (a) y (c), se puede notar que los cambios entre corridas tienen saltos menos pronunciados para las mediciones con los features seleccionados más importantes (es decir, son más estables). En ésas visualizaciones los modelos con features seleccionadas

lograron un menor desvío que los que usaron todas las columnas cuando se aumentó la cantidad del hiper parámetro correspondiente.

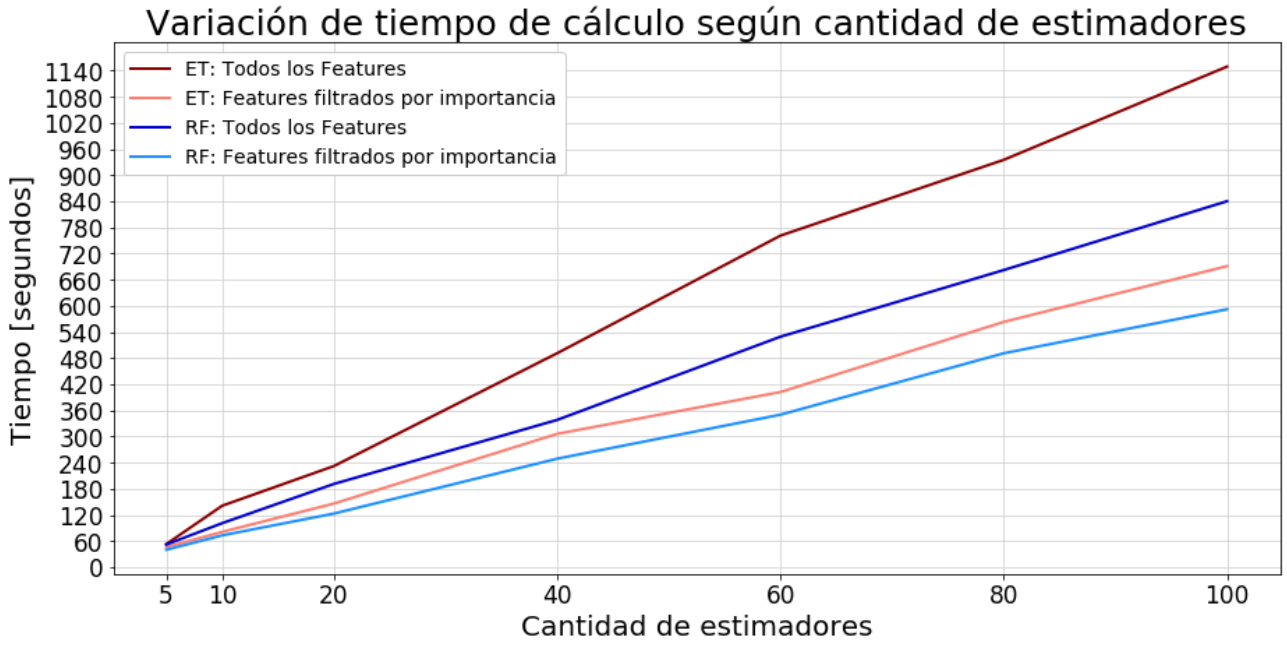

Fig. 4.15.

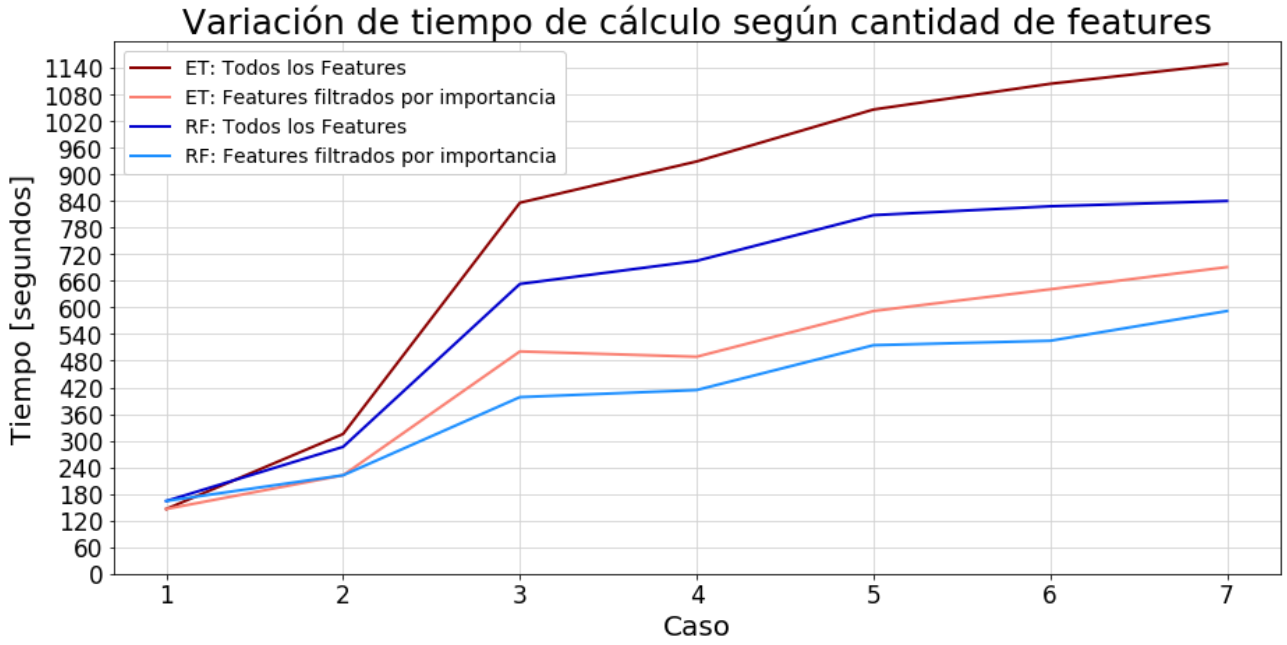

Fig. 4.16.

Por lo visto en las figuras de variación de errores, se puede reducir la cantidad de columnas con una penalización baja en la predicción si se seleccionan adecuadamente. Aquí se observa que también se puede reducir el tiempo de cálculo.

En la figura superior se ve un aumento de tendencia lineal para un aumento de estimadores a usar para el entrenamiento. La variación (pendiente) es más pronunciada cuando se usan todos los features para ambos algoritmos.

4.7. XGBoost

La implementación del modelo anterior Extreme Gradient Boosting utiliza derivadas de segundo orden para mejorar la información de optimización en la reducción del error, y tiene la ventaja de poderse paralelizar, lo cual no solo alienta a realizar más pruebas con él dado el tiempo disponible, sino que incita a un tunning de features más exacerbado bajo la premisa de que se podrá visualizar en un tiempo razonable el resultado, gracias al uso de varios threads.

Desde un comienzo, XGBoost tuvo un buen rendimiento con los datos pre procesados. Solo a partir de los parámetros por defecto se obtuvo un error promedio de menos de 800 000, algo no logrado con ningún otro modelo sin el uso de los features antes descritos. Con un poco de tunning de los hiper parámetros, se logró una subida en Kaggle con un error promedio de menos de 690 000, que en su momento fue la mejor ejecución lograda. Agregando los features descritos en la sección de feature engineering, se bajo el error promedio a menos de 509 000, superando al modelo de los árboles randomizados.

Algo interesante que se observó fue que agregar las descripciones procesadas mediante THT no mejoraba casi nada el resultado, en contraste con LightGBM. Si no fuera porque bajo estas mismas condiciones LightGBM se comportó mejor, XGBoost hubiera sido la opción predilecta para continuar con un tunning aún más detallado. Pero el último modelo de boosting a presentar en la próxima sección, tuvo predicciones aún mejores, aprovechando aún más features que cualquiera de los modelos anteriores, y se convirtió en la opción selecta para las predicciones.

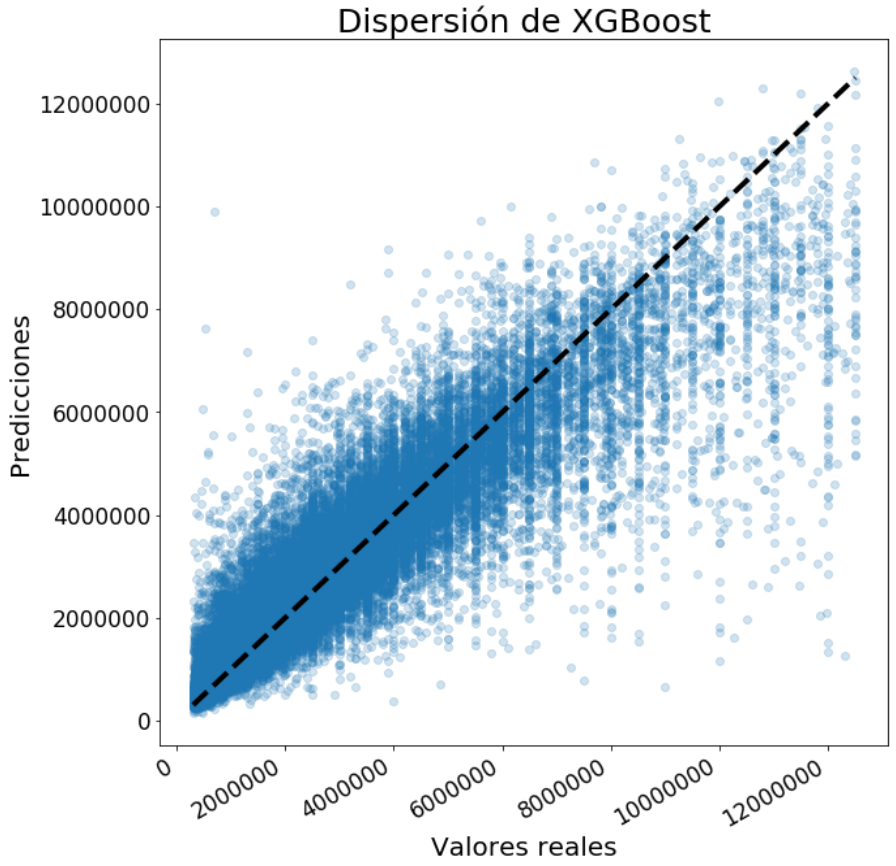

Fig. 4.17.

4.8. Light GBM

LightGBM es un algoritmo de gradient boosting mantenido por Microsoft que difiere en su contraparte XGBoost en la implementación, y en algunas técnicas que utiliza para mejorar el rendimiento, como Gradient-based One-Side Sampling (para no descartar instancias de predicción con valores bajos de

gradiente, evitando alterar por este medio la distribución de los datos de entrenamiento) y Exclusive Feature Bundling (acelera la velocidad de las predicciones identificando los features mutuamente excluyentes y considerándolos dentro de un mismo bundle).

Desde un principio se obtuvieron muy buenos resultados con LGBM. En particular, utilizando los hiper parámetros por defecto con los features mencionados se logró un error promedio de 632 550, un rendimiento muy superior a cualquiera de los otros modelos mencionados en las mismas condiciones. Tanto en este caso, como a partir un tunning manual, los resultados de este modelo superaron a los de XGBoost, convirtiéndose en el algoritmo que mejor predicciones otorgó. Esto probablemente se deba a que los algoritmos de boosting como XGBoost puedan considerarse como el estado del arte para este tipo de predicciones, y Light GBM en particular es una versión mejorada de la implementación del mismo.

No solo el buen score fue motivante para profundizar en el ajuste de hiper parámetros desde el principio, sino que también el poco tiempo de ejecución que tomó obtener los resultados (aún comparando contra XGBoost) fueron un motor para propulsar las pruebas sobre el modelo.

Tan sólo con incrementar el número de iteraciones a 4 000 se consiguió una mejora de más de 100 000 sobre las predicciones, obteniendo un error promedio de 528 844; y aumentar aún más iteraciones trajo consigo mejoras aún mayores. El aumento del número de iteraciones también llevó consigo un aumento apreciable en el tiempo de ejecución del algoritmo. Este hiper parámetro resultó de mucha importancia, dado que cuando se quería mejorar el error promedio en muchos casos simpemente bastaba con incrementar el mismo, utilizando un valor alto en conjunto con un learning rate bajo. Por supuesto, siempre se llegaba a un punto (cercano a las 40 000 iteraciones) donde continuar aumentando el número sólo aportaba mejoras despreciables, mientras que aumentaban desproporcionadamente el tiempo de procesamiento, pero siendo el mismo en todos los casos inferior a dos horas (otro aspecto a destacar, comparado con otros modelos descritos que en algunos casos no parecían terminar nunca el entrenamiento).

Asociado a la cantidad de iteraciones está el *learning_rate*, el cual regula qué tan rápido "aprende" el algoritmo. En términos de gradient boosting, es el factor que suaviza las predicciones de los árboles para que no se prediga todo en la primer iteración. Para este hiper parámetro se encontró que valores entre 0.1 y 0.01 resultaban adecuados.

Por otro lado aumentar el número de hojas de los árboles ("num_leaves") en la mayoría de los casos también resultó ventajoso; pero dicho aumento solo se dio hasta cierto punto, pues como se recomienda en la documentación de LGBM, se debe manejar este parámetro con cuidado debido a que es muy sensible al overfitting. Los valores que dieron los mejores resultados estuvieron en el orden de los 300.

El objetivo (parámetro "objective") osciló entre el default ("regression") y el objetivo "regression_l1", el cual es un objetivo que busca minimizar el MAE. A grandes rasgos, este hiper parámetro resultó un tanto indiferente, dado que ambos objetivos daban valores de errores promedio similares, y parecían responder de manera semejante al cambio de los otros hiper parámetros. A pesar de esto, también se probaron otros objetivos (que no fueron tan fructíferos): "poisson", "quantile", "tweedie" y "gamma".

Finalmente, se destaca el hiper parámetro "max_bin", el cual se encarga de decidir en cuántos buckets se discretizan las variables que son numéricas. Aumentar el parámetro trajo consigo una mejora en las predicciones, hasta cierto punto (por ejemplo, pasar de 2 000 a 10 000 realmente no impactaba significativamente), además de aumentar considerablemente el consumo de memoria y el tiempo de entrenamiento. En general, lo que mejor resultados dió fue utilizar valores en el rango de 1 500 a 2 000.

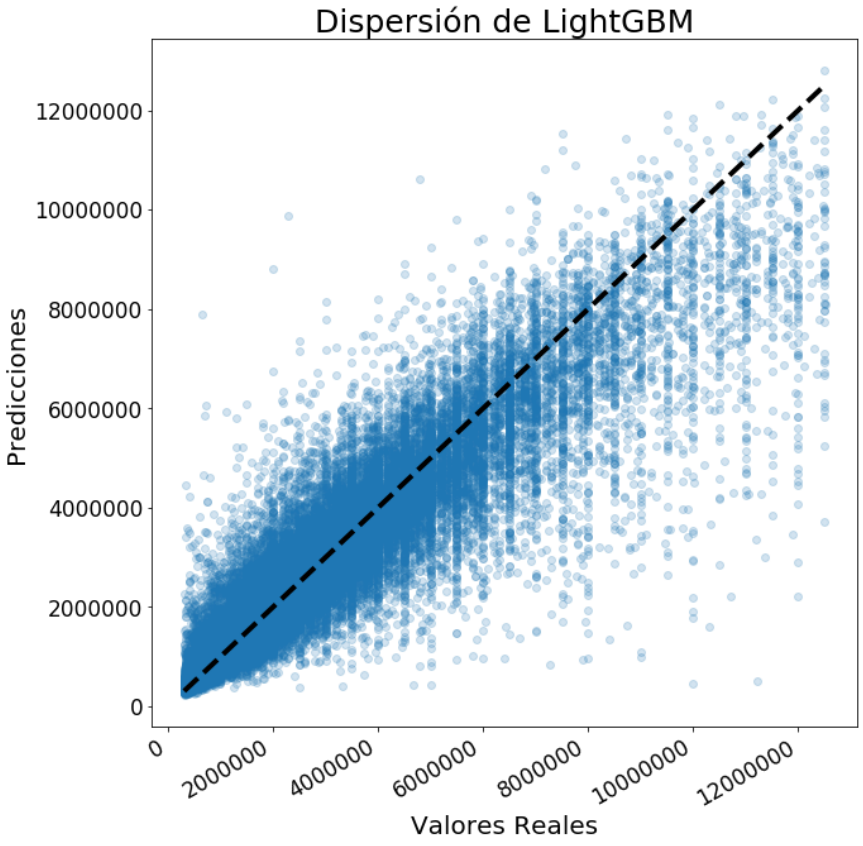

Fig. 4.18.

4.9. Ensambles

A parte de los algoritmos de boosting, se probó realizar otros tipos de ensambles entre los algoritmos descritos anteriormente.

En primer lugar se probó combinar un clasificador multi-clase con un algoritmo de regresión mediante stacking/blending de la siguiente manera:

- se partía a los precios en N buckets, utilizando para esto los valores de distintos cuantiles: se creaba un bucket para las propiedades que estaban entre el cuantil 0.1 y el 0.2, otro para los que están entre 0.2 y 0.3 y así sucesivamente;
- se consideraba a cada uno de los buckets como una clase distinta y se le asignaba un número;
- se le asignaba a cada propiedad como target el número de bucket al que pertenece;
- se utilizaba algún algoritmo de clasificación para predecir las clases de cada propiedad;
- se usa la clase predicha para cada propiedad como feature.

La idea surgió a partir de lo visto en clase en la charla que dio el equipo de Navent y luego de leer una variedad de artículos online y notebooks de Kaggle, en especial relacionados con la competencia sobre sugerencia de precios de artículos de Mercari[1]. En ellos se describía el proceso anterior, y se sugería el uso de una red neuronal como algoritmo de clasificación, y dado que la red neuronal con la que cuenta sklearn no es muy configurable y es bastante lenta, se decidió crear una red propia mediante la librería Keras y, utilizar el servicio de Google Colab para entrenar la red. La ventaja de esto es que Colab permite entrenar la red utilizando los llamados TPU (Tensor Processing Unit) de Google, que son unidades de procesamiento especialmente diseñadas para entrenar modelos de machine learning, de forma que el entrenamiento de la red se acelera considerablemente.

[1] https://www.kaggle.com/c/mercari-price-suggestion-challenge/overview

Se probaron varias configuraciones para la red (distinta cantidad de capas, cantidad de nodos para las capas, funciones de activación, funciones de optimización) pero no se pudo lograr un buen resultado de forma preliminar con la misma. En el tiempo asignado a esto, resultó complicado entender la interfaz y el funcionamiento de la librería. Posiblemente, teniendo mayor conocimiento en el área, y con un poco más de tiempo y práctica se podría haber logrado un buen resultado.

El mejor score que se obtuvo fue un 0.15 de accuracy (esto es, se clasificaron bien a un 15% de las muestras), por lo que se decidió probar otros modelos para la clasificación. La elección final fue KNN, que obtuvo un score de 0.189. Luego de esto se utilizaron las predicciones del algoritmo, y se agregaron como feature al dataset, utilizando como algoritmo blender a LightGBM (para la regresión). Los resultados fueron variados: se observó que al agregar este feature las predicciones del algoritmo con los parámetros por defecto mejoraba, pero se volvía un tanto invariante frente al tunning de hiper parámetros. Es decir, variar los parámetros del algoritmo ya no hacía que las predicciones (y por lo tanto el score) variaran de la misma manera. Con los parámetros por defecto se obtuvo un resultado de 533 000 que, comparado con los 644 000 que se obtenían sin este feature, se considera una gran mejora. Al momento del tunning, se observó que no variaba mucho el resultado, o incluso en algunos casos empeoraba, solo logrando que las predicciones bajen a 520 000.

Si se hubiese dispuesto de más tiempo para este modelo de ensambles en particular, se podrían haber explorado más variantes, distintos algoritmos de clasificación y regresión, y se podría haber probado un tuneo automatizado de parámetros para el blender. Por otra parte, se le podría haber dedicado más tiempo a la red neuronal, a entender bien el funcionamiento de la misma y aprender a construir un modelo más robusto, para no terminar por tomar la decisión "rápida" de utilizar el modelo de KNN como clasificador y LGBM como regresor

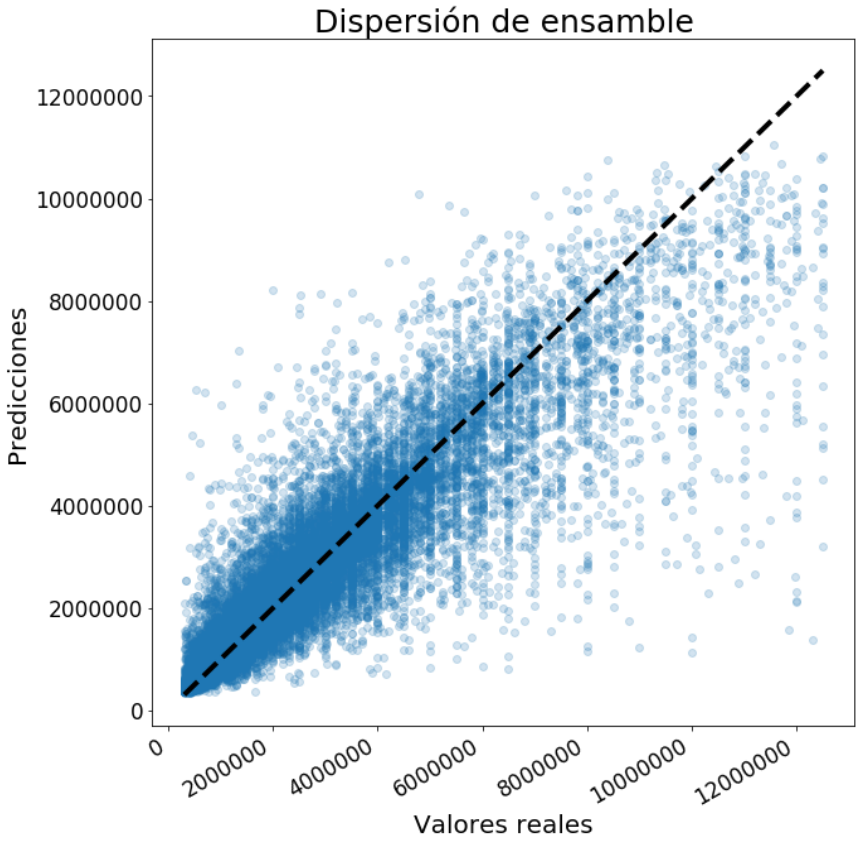

Fig. 4.19.

4.10. Resumen

Se sintetiza brevemente la comparación de las ejecuciones de referencia de cada uno de los modelos

a través del siguiente gráfico, con el objetivo de presentar de forma resumida un contraste entre los mismos.

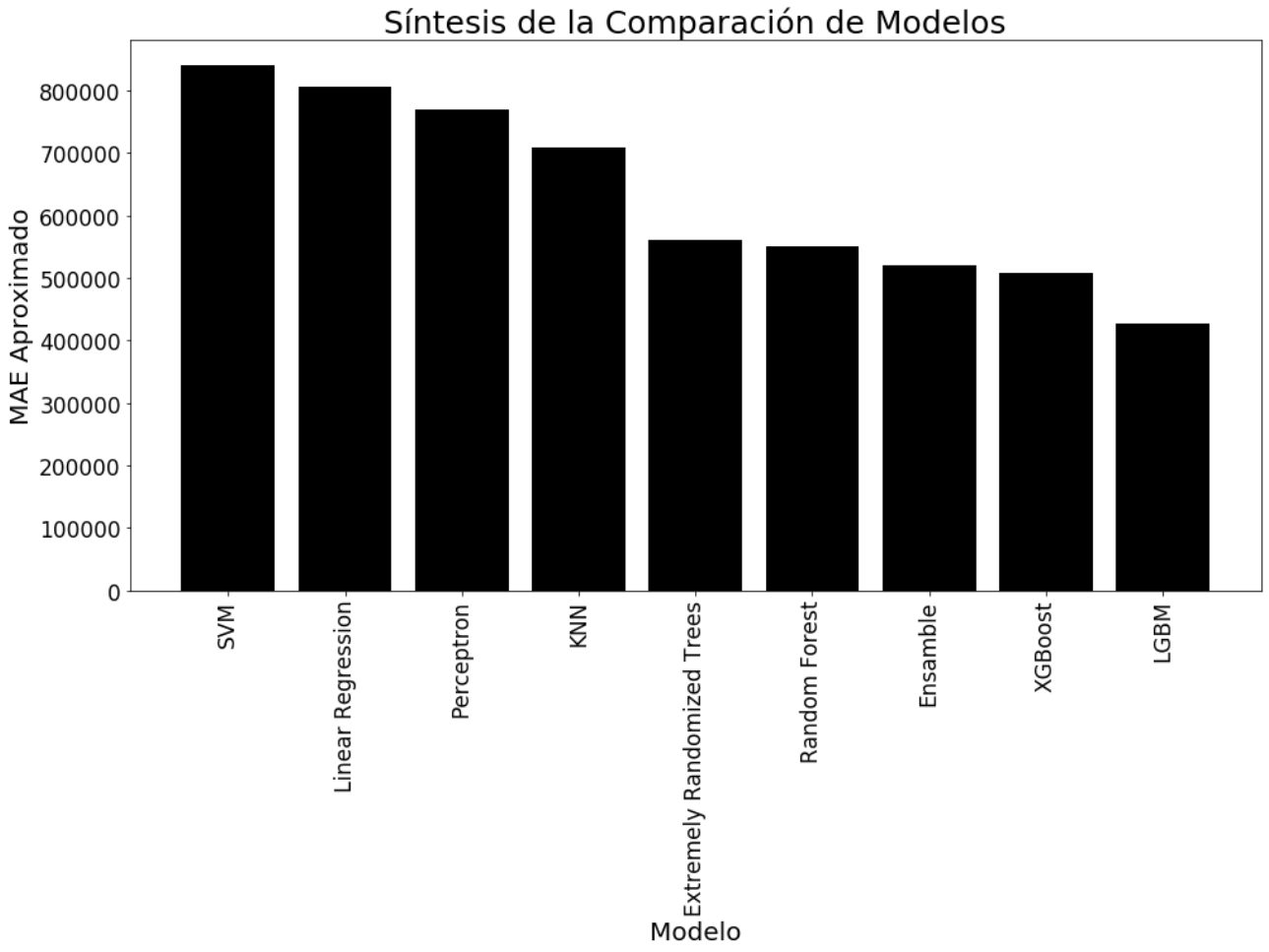

Fig. 4.20.

Se aprecia que todo el esfuerzo hecho para el tunning y el agregado de features del modelo final (que se describirán en la sección anterior) logra prácticamente reducir a la mitad el error promedio de la peor predicción, mientras que resulta interesante que la misma carga de trabajo no parece lograr una mejora tan significativa con respecto a los modelos de árboles aleatorios (que utilizaron otro conjunto de features). Sin embargo, de cara al problema de referencia se entiende que cada mejora lograda, al menos dentro de las primeras cuatro cifras significativas, realmente tiene un peso de interés tanto con respecto a la calidad de las predicciones, como para con la competencia de Kaggle.

5. Algoritmo Final

El algoritmo final, y al que se le decidió dedicar la mayor parte del tiempo y el esfuerzo fue LightGBM, por varios motivos:

- los buenos resultados que se obtuvieron desde un principio;
- la gran velocidad de entrenamiento del mismo, incluso con una gran cantidad de features;
- la gran cantidad de hiper-parámetros para configurar, y todas las posibilidades de mejora que estos representaban;
- la capacidad de manejar variables categóricas nativamente (sin necesidad de OHE, por ejemplo) en comparación con XGBoost, y su posibilidad de manejar los NaN de forma nativa.

En las subsecciones posteriores se profundiza en el uso del modelo, y se explican variantes al feature engineering realizadas para el submit final a Kaggle.

5.1. Hiper Parámetros

Se probaron muchas configuraciones sobre los hiper parámetros manualmente a medida que se testeaban features nuevos, teniendo siempre en mente el riesgo de overfitting, por lo que ocasionalmente se variaba el seed del Train-Test-Split, para generar los sets de entrenamiento y de test. Adicionalmente, también se incluyó un K-Fold Cross Validation para ver si había una diferencia significativa entre los resultados que se estaban obteniendo con el promedio de los folds, y el split particular que se realizaba para entrenar.

A parte del tunning manual que se realizaba de manera continua sobre el algoritmo, también se utilizaron librerías dedicadas al ajuste de hiper parámetros automatizado. Para explorar el amplio espacio de posibilidades y de configuraciones de valores para los parámetros se utilizó la librería Hyperopt, la cual permite optimizar hiper parámetros en forma inteligente utilizando lo que se conoce como "Estimadores de Parzen" estructurados en árboles. En términos simples, el algoritmo elige el siguiente grupo de hiper parámetros a testear teniendo en cuenta todos los anteriores parámetros que ya fueron probados, por lo que se espera que el algoritmo en cierta forma este "refinando" los hiper parámetros seleccionados iterativamente. La principal ventaja de la librería es que permite el uso de rangos continuos para la búsqueda de los hiper parámetros, en contraste con el uso de una lista numerable de valores a probar (como se haría en Grid Search por ejemplo). Adicionalmente, Hyperopt ofrece la posibilidad de serializar la información aprendida una vez esté completada una cierta cantidad de iteraciones de búsqueda de hiper parámetros, de forma que se puede volver a cargar más tarde, y continuar tuneando parámetros desde donde se lo dejo.

Como base, se realizó un análisis de sensibilidad de los hiper parámetros para tener un panorama de cuáles incluir, cuáles eran los que mejor funcionaban, y particularmente cuáles eran los rangos de valores a utilizar al momento de hacer el tunning automatizado. Para tunear los hiper-parámetros del algoritmo, se utilizaron 150 iteraciones de refinamiento, cada una con Cross validation de 4 folds. Dado que esto conlleva una gran cantidad de corridas de entrenamiento y validación (600), se tuvo que limitar el rango de algunos hiper parámetros de entrenamiento para que el tunning termine en un tiempo razonable.Por ejemplo la cantidad de iteraciones fue limitada al rango [100, 2000], la cantidad de hojas posibles fue limitada superiormente a 300, y así sucesivamente con otros hiper parámetros que se consideraban como riesgosos para aumentar demasiado el tiempo de ejecución.

En el proceso anterior no se utilizaron las descripciones, dado que esto también aumenta el tiempo de entrenamiento considerablemente y en algunos casos (dependiendo de la capacidad de la PC) ocasionaba un uso excesivo de RAM.

El proceso duró aproximadamente 8 horas y media y se obtuvieron los siguiente resultados:

- feature_fraction: 0.5959853966635414
- lambda_l1: 26.82894063964651
- learning_rate: 0.07141681335686303

- max_bin: 4600
- min_data_in_leaf: 20
- num_iterations: 1900
- num_leaves: 250

A pesar de no haber incluido las descripciones vectorizadas al momento de tunear, se pudo comprobar que luego de agregarlas, los valores de hiper parámetros obtenidos se ajustaban bastante bien al set de datos. De todas formas, se entiende que los hiper parámetros óptimos podrían ser distintos luego de agregar este feature.

Para la corrida final se modificaron ligeramente algunos hiper parámetros, utilizando valores que se sabía que iban a dar un mejor resultado, como por ejemplo aumentar la cantidad de iteraciones a medida que se disminuye la tasa de aprendizaje. Adicionalmente, se aumentó también la penalización "lambda_l1", dada la observación de que mejoraba ligeramente el score que se obtenía mediante Cross Validation, lo cual coincide con el objetivo del parámetro que es el de penalizar modelos muy complejos, y así prevenir el overfitting.

Los valores de los hiper parámetros finales entonces terminaron siendo:

- feature_fraction: 0.5959853966635414
- lambda_l1: 100.82894063964651
- learning_rate: 0.01141681335686303
- max_bin: 1500
- min_data_in_leaf: 20
- num_iterations: 50000
- num_leaves: 270

Con esta configuración y los features mencionados anteriormente se logró un error promedio en la subida a Kaggle de 427 000. Cabe destacar que si se hubieran utilizado 40 000 iteraciones, el resultado hubiera sido prácticamente el mismo: 427 300, remarcando nuevamente lo mencionado sobre el límite de la cantidad de iteraciones como optimizador de las predicciones por sí solo.

5.2 Features adicionales

Para la mejor corrida de predicciones antes mencionada, se utilizaron algunos features adicionales, o bien, features con un procesamiento distinto al descrito anteriormente. Los mismos se describen a continuación.

5.2.1. Tratamiento de columnas categóricas

LightGBM tiene la capacidad de manejar columnas categorícas sin la necesidad de tener que aplicar OHE, Binary Encoding o encodings similares; solo se deben convertir las columnas categóricas de forma que cada categoría quede representada por un entero (esto se puede lograr fácilmente con el LabelEncoder de sklearn) y especificar al momento de entrenar cuáles columnas se quieren tratar como categóricas. Utilizando esta funcionalidad del algoritmo para las columnas: *provincia*, *ciudad* (no incluida anteriormente), *idzona*, *tipo de propiedad*, y todas las columnas descriptivas mencionadas en la 3.2.3, se obtuvo una mejora considerable respecto al encoding que se estaba utilizando antes.

Otra ventaja que esto representó fue un gran ahorro de memoria, dado que no se tenía que lidiar con cientos de columnas para representar provincias, ciudades y etc. Además, permitió tratar a idzona como una variable categórica, cosa que usando One Hot Encoding (por ejemplo) hubiera sido totalmente inviable dado la enorme cantidad de valores diferentes que hay para dicha columna.

5.2.2. Tratamiento de NANs y valores faltantes

LightGBM tiene la capacidad de manejar columnas con NANs. Lo que hace cuando se encuentra con un valor faltante al momento de splitear es simplemente ignorar el NAN, y colocar el dato en el bucket en el cual la pérdida (loss) se vea minimizada. Las corridas que mejor resultado dieron fueron aquellas hechas sin rellenar los NANs presentes en el set de datos: se deja que LightGBM maneje estos valores faltantes de

manera inteligente. Esto permitió volver a incluir columnas que se habían desechado inicialmente: *latitud* y la *longitud*.

5.2.3. Suma de metros totales y cubiertos

Se agregó un feature que simplemente era la suma de los metros totales y cubiertos. En caso de que uno de los dos estuviera ausente, se rellenaba con el otro. Esto trajo una pequeña mejora sobre el score final.

5.2.4. Transformación del precio

A diferencia de lo ocurrido con otros modelos, para la predicción del precio no se utilizó la transformación del logaritmo para el precio, sino la de la raíz cuadrada (ver 3.2.1), a partir del hiper parámetro "reg_sqrt".

5.2.5. THT de títulos ampliado

Para lograr una pequeña mejora adicional, el Hashing Trick aplicado sobre los títulos fue utilizado con una vectorización más amplia, aplicando el mismo con un vector de 200 componentes en vez de 50.

5.3. Feature Importance

Para poder visualizar y explicar un poco más en detalle la importancia de los features para el algoritmo, se utilizó la librería Shap (SHapley Additive exPlanations) cuyo principal objetivo es permitir explicar la salida de los modelos de machine learning, a partir de gráficos detallados, y mostrar de forma interactiva el peso que tiene cada feature en la salida. Dicho peso lo cuantifica con los llamados Shap values.

En el siguiente gráfico se muestra el feature importance de LightGBM:

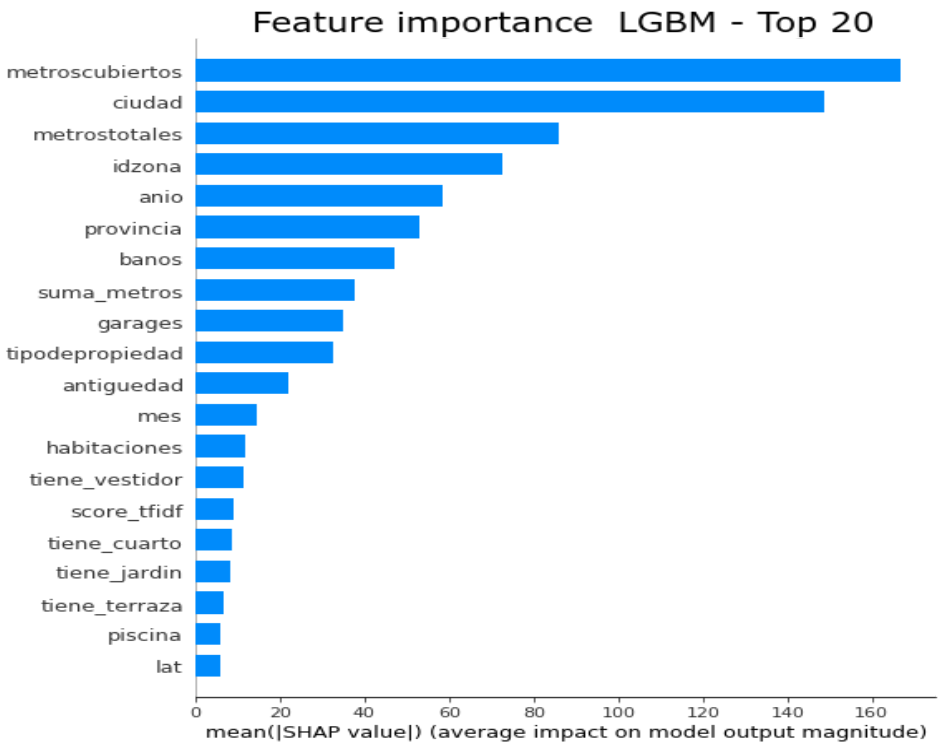

Fig. 5.1.

También podemos ver, además de su importancia, cómo es que cada feature se relaciona con la variable de salida:

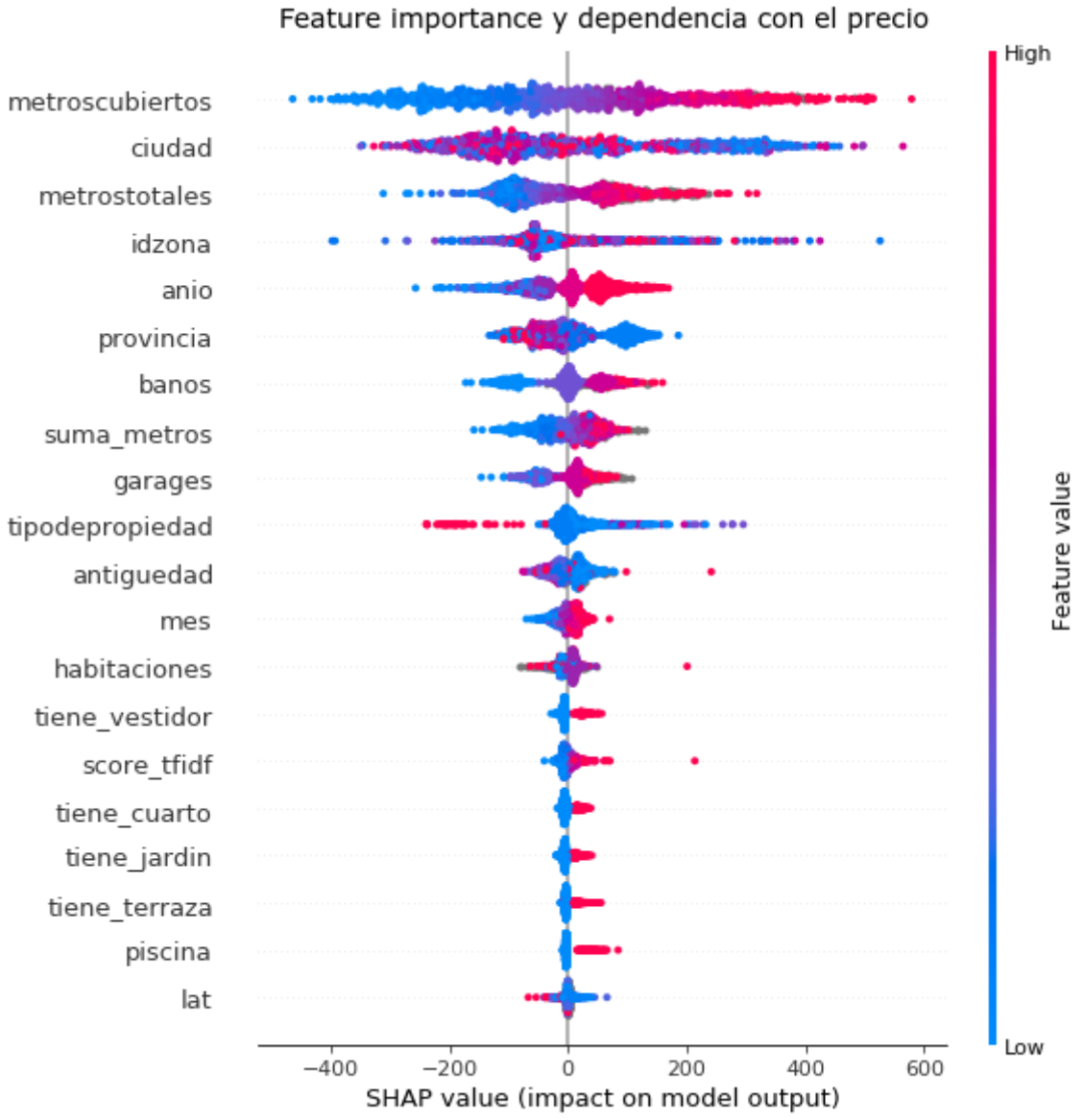

Fig. 5.2.

En el gráfico se puede ver, por ejemplo, que a medida que aumentan los metros cuadrados, aumenta el precio predicho, y que en el caso de la antigüedad parece pasar lo contrario: valores altos hacen que la predicción disminuya. Para las variables categóricas, como ciudad y provincia, no tiene mucho sentido la interpretación del gráfico, dado que simplemente el modelo está usando encoding arbitrario para representar a las ciudades, por lo que no pueden hacerse asociaciones del estilo "a mayor valor de ciudad mayor precio". En particular, en el gráfico correspondiente a las ciudades se puede ver que es un gráfico disperso, sin una clara separación.

En los dos gráficos anteriores se puede ver que la cantidad de metros cubiertos y totales tienen una muy alta correlación con el precio, lo cual se había observado en el trabajo práctico anterior. Con Shap, se puede ver como el modelo interpreta a un feature en particular y cómo este afecta la salida del mismo. Se muestra a continuación la relación de los metros cubiertos con el precio:

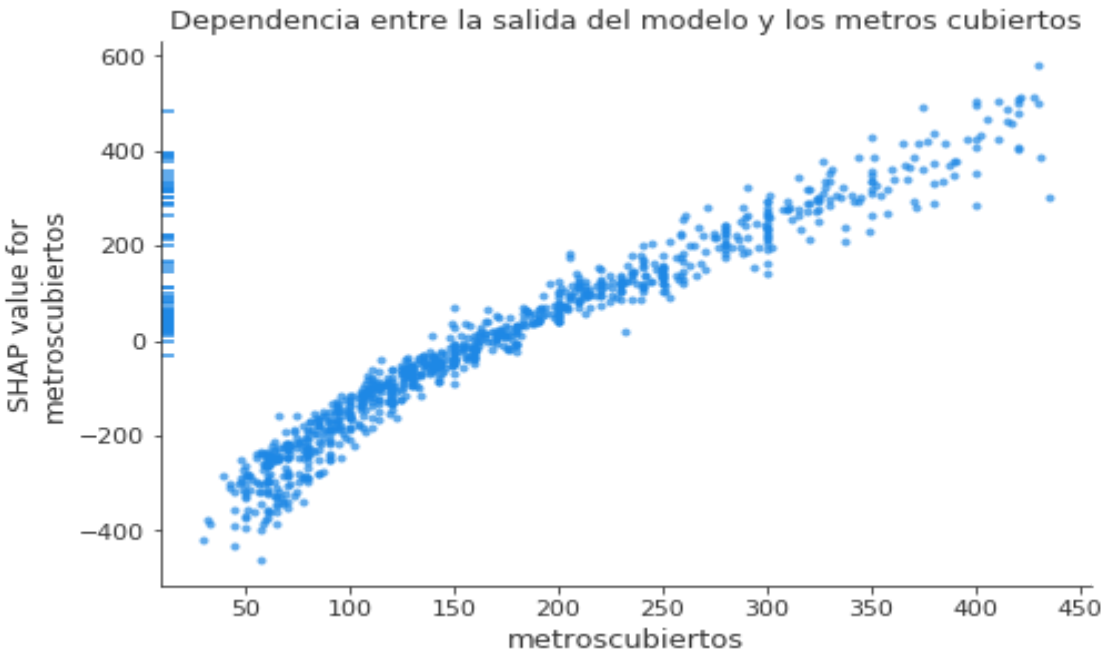

Fig. 5.3.

Si comparamos el gráfico con lo la visualización obtenida en el trabajo práctico anterior:

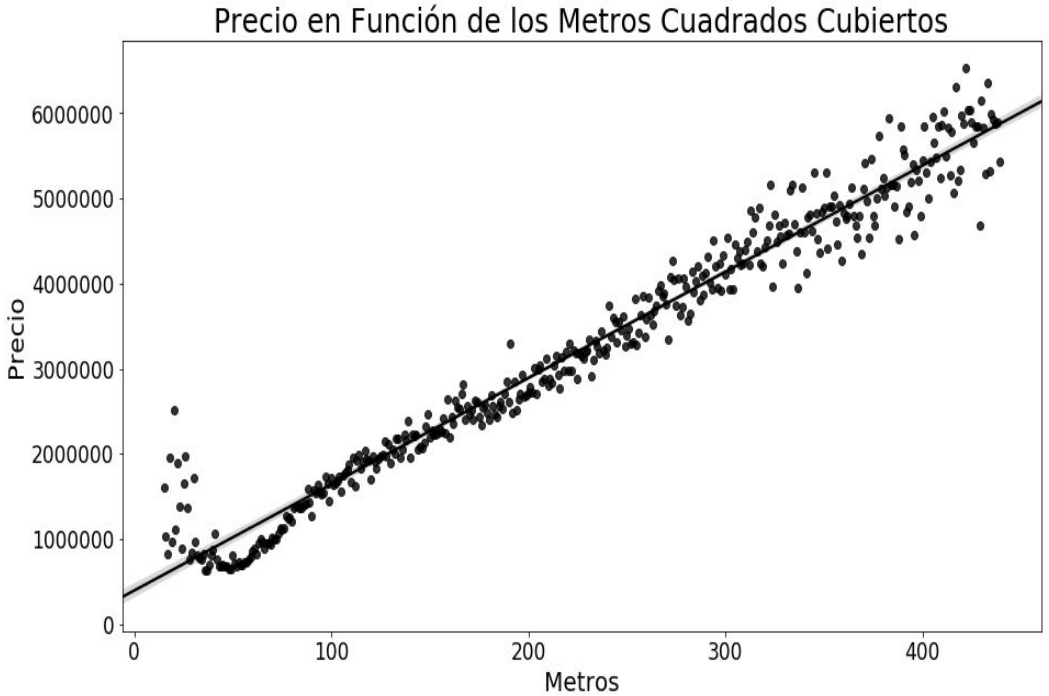

Fig. 5.4.

se puede concluir que el algoritmo aprendió muy bien la relación entre estas dos variables.

6. Conclusiones

El modelo de boosting fue el que mejor desempeño tuvo para las predicciones, incluso cuando sólo se habían utilizado los features nativos del dataset. Es conocido que el modelo de XGBoost y otras variantes asociadas, como por ejemplo el distinguido Light GBM, pueden ser considerados como el estado del arte en

predicciones de datos estructurados. Para el presente trabajo práctico parece haberse cumplido el pronóstico, ya que los algoritmos basados en boosting con aprendizaje a partir de árboles fueron los que dominaron en todo momento las predicciones, y con lo que mayores mejoras se conseguían al implementar nuevos features que terminaron siendo exitosos.

Se entiende que las predicciones logradas son de buena calidad, incluso bajo la consideración de que sólo se tienen conocimientos fundacionales sobre Machine Learning, y contemplando que se cuenta solo con esta experiencia en el framework de sklearn. Se tiene un modelo que predice con un error promedio de menos de $ 450 000, que probablemente sea un nivel de exactitud comparable al de una persona adecuada que realiza la tasación con la misma información disponible, ya que con un precio promedio de 2,5 millones, se está hablando de un error de menos del 20%.

Si se tiene en cuenta adicionalmente que el set de datos era abundante (aunque no masivo), se observa como consecuencia de su nivel de representatividad asociado que se pudo lograr entrenar un modelo que puede servir de referencia para estimar el precio de una propiedad fehacientemente para los datos con los que se cuenta. Más aún: bajo el supuesto de que una propiedad no sea "de las más caras", el modelo tendrá una cota de error bastante menor, ya que como se pudo apreciar en los gráficos de cada algoritmo, el mayor peso que toma el error absoluto promedio se da por las peores estimaciones de las propiedades de mayor precio, las cuales al ser la minoría son estimadas con menor precisión.

Del análisis de los algoritmos de Random Forest y Extremely Randomized Trees, queda claro que aumentar las cantidades de estimadores y features mejora las predicciones. Pero la forma en que lo hacen por separado es diferente, y una síntesis de esto puede ayudar a reforzar los conceptos bajo estudio.

En el análisis que se realizó variando el número de estimadores siempre se utilizó la misma cantidad de features. Esa es la máxima cantidad de features destinadas para el otro análisis (variación de features). Al aumentar los estimadores se llega a un tope de exactitud: la meseta en la mejora de la predicción. De esto se puede interpretar que con una cantidad fija de información (los features), el resultado tendrá un límite de cuánto se puede aproximar al valor a predecir. Esto se nota en la estabilización del error y el score, (fig. 4.10 y fig. 4.12). También se observa una estabilización en los desvíos de ambas cantidades (fig. 4.14 (a) y fig. 4.14 (b)).

En el análisis de variación de features se mantuvo constante la cantidad de árboles. Se inició con pocos atributos y se fueron agregando más en casos de prueba subsiguientes. Desde el primer caso de estudio se puede ver que el error es superior al del primer caso en el análisis anterior (todas las columnas y una cantidad mínima de 5 árboles). De esto se puede inferir que, usando una cantidad fija de estimadores, agregar features incorpora información al modelo. A lo largo de las pruebas el error disminuye y el score aumenta (fig. 4.11 y fig. 4.13). En cuanto a los desvíos de estas cantidades se logra una estabilización y, a diferencia del análisis anterior, se tiene también una disminución de los mismos (fig. 4.14 (c) y fig. 4.14 (d)).

En cuanto a sugerencias para Navent, se recomienda la inclusión de los atributos descriptivos minados a partir del título y la descripción dentro de las características de las propiedades ofertadas, los cuales parecen representativos de cualidades de la propiedad. Esto puede hacerse de forma automática, sin necesidad de que el usuario tenga que (por ejemplo) tildar boxes con características especiales. Adicionalmente, para la mejora de las predicciones podría ser interesante la inclusión de imágenes de las distintas publicaciones, ya que si se procesan de forma adecuada pueden mejorar mucho la calidad de las predicciones, siendo la impresión visual del inmueble (tanto exterior como interior) un componente clave de la tasación. Esto último a su vez revitalizará la intención del uso de las redes neuronales como parte del modelo para las predicciones de los precios, que en la solución presentada fueron dejadas de lado en oposición a los modelos de árboles.