Universidad De Buenos Aires
Facultad De Ingeniería

Año 2019 - 2do Cuatrimestre

Organización de Datos (75.06)

Trabajo Práctico Nro. 2
Predicción del precio de una publicación

Integrantes - Grupo 1 - "Organización de Gatos"

- **Harfuch, Mateo - 95049**

Índice de contenidos

Índice de contenidos **2**

1. Introducción **3**

2. Metodología de Trabajo **4**

2.1 Recursos y entorno 4

2.2 Pipeline de trabajo 4

2.3 Arquitectura de la solución 5

3. Feature engineering **6**

4. Modelos **6**

4.1 División de set de datos y métrica 6

4.2 Modelo PromedioZona 7

4.3 Modelo RegresionLineal 7

4.4 Modelo XGBoostRegressor 7

4.5 Modelo XGBoostClassifier y predicción de faltantes 8

4.6 Modelo MLP_Regressor 9

5. Ensambles **9**

5.1 Ensamble por pesos 9

5.2 Ensamble por concatenación - Mejor puntaje 10

6. Conclusiones **10**

1. Introducción

A partir del análisis exploratorio llevado a cabo en el primer trabajo práctico, implementamos un algoritmo que permite predecir el precio de una publicación. Para lograrlo, utilizamos un set de datos proporcionado por Navent y diversas técnicas de procesamiento de datos y machine learning.

El problema planteado es de regresión ya que el target (precio) es un valor numérico sobre un continuo. Por otro lado, establecimos como hipótesis que, dado un set de datos rico, no es un problema difícil por lo que esperábamos buenos resultados.

La solución propuesta participó de una competencia en Kaggle[1] donde distintos grupos subieron sus resultados y a la fecha de este informe faltando 4 días para el cierre ranqueaba en el puesto 18 de 37, con puntaje 530k.

En cuanto a las herramientas y el entorno de trabajo, se usaron distintas librerías de Python trabajando sobre notebooks de Jupyter. El código desarrollado se encuentra alojado en un repositorio público en Github[2] y en este documento se volcaron las ideas sobre las cuales se trabajó para llegar al algoritmo final.

[1] https://www.kaggle.com/c/Inmuebles24/
[2] https://github.com/unmateo/7506-TP

2. Metodología de Trabajo

2.1 Recursos y entorno

Todo el trabajo se desarrolló en un entorno local corriendo en una notebook con procesador Intel I5 de ocho núcleos y 15 GiB de memoria RAM disponibles sobre un sistema operativo Ubuntu. En términos generales estos recursos fueron suficientes, pero difícilmente pueda replicarse el trabajo con menor disponibilidad ya que en algunos casos se trabajó al límite de la capacidad.

Para manejar las dependencias se utilizó Pipenv, por lo que las librerías y versiones utilizadas están listadas en el Pipfile del repositorio y replicar el ambiente no debería ser un problema mayor.

El código está disponible en sus versiones notebook, html y py, gracias a un hook instalado para que se haga el guardado automático en los tres tipos de archivo y esto nos permitió poder llevar un mejor control de cambios.

2.2 Pipeline de trabajo

Para agilizar el desarrollo y llevar un control sobre las pruebas que se fueron haciendo, se respetó el siguiente pipeline de trabajo:

- Desarrollo de un modelo
- Validación local
- Submit en la competencia
- Registro del resultado
- Commit del código

Todos los modelos desarrollados están bajo el directorio modelos. Las predicciones que se subieron a la competencia están bajo el directorio predicciones y en el archivo resultados.csv se registró cada una de ellas con el puntaje obtenido y un comentario de referencia.

2.3 Arquitectura de la solución

Con el objetivo de respetar el pipeline y reutilizar la mayor cantidad de código posible se armaron los módulos modelo.py y datos.py para agrupar el comportamiento que iba a compartirse entre los distintos modelos.

En datos.py se encuentra toda la lógica de features y procesamiento del set de datos original. Dentro de este módulo, la función levantar_datos disponibiliza el .csv como un dataframe de pandas con tipos optimizados, ya dividido en train, test y submit. Esta función puede recibir además como parámetro el listados de features que se requiere. De esta forma todos los modelos tienen acceso a todos los features, pero cada uno puede “pedir” sólo los que necesita.

Por otro lado, en modelo.py se definió la clase *Modelo* de la cual heredan todos los modelos desarrollados. Esta clase define una interfaz común y proporciona los métodos cuya implementación es igual para todos los modelos. Estos métodos son:

- cargar_datos: llama a levantar_datos y setea propiedades
- puntuar: recibe una serie de predicciones y sus valores reales y calcula el puntaje según la métrica de la competencia
- validar: predice set de test y devuelve el puntaje obtenido
- presentar: recibe predicciones, las guarda, hace el submit en la competencia a través de la API de kaggle y registra el resultado.

La única funcionalidad común que no se desarrolló es el tuning de hiperparámetros, ya que cuando surgió la necesidad casi todos los modelos ya estaban desarrollados bajo esta interfaz y adaptarla a la API de sklearn (fit, transform, predict) hubiera llevado más tiempo del que se tenía disponible. Por lo tanto, en los casos en los que correspondía hacer tuning se hizo individualmente.

3. Feature engineering

Siendo que en primer trabajo práctico se hizo un análisis exhaustivo de la información que teníamos, sólo mencionaremos los features que agregamos y un resumen de cómo se trabajó con los datos.

Respecto a los features, además de los que ya teníamos disponibles agregamos:

- Temporales: dividiendo fecha en año, mes y día
- Dólar: usando un set de datos externo, con la cotización para cada día
- NPL: procesando los campos título y descripción pudimos utilizar esa información cuantitativa y cualitativamente.

Cabe mencionar que los datos de geolocalización (latitud y longitud) no se utilizaron sólo porque procesarlos implicaba demasiado tiempo, pero creemos que de haberlo hecho probablemente hubieran aportado información valiosa.

Basándonos en el análisis del anterior trabajo, determinamos que podíamos confiar en los datos que teníamos y que no valía la pena invertir tiempo en “limpiarlos”. Sin embargo, no descartamos que en un análisis más profundo se pueda trabajar este tópico y conseguir mejores resultados.

Por último, lo que sí se hizo fue trabajar la carga de los datos eligiendo los tipos óptimos para cada caso y de esta forma liberar recursos para que puedan ser utilizados en mejorar el entrenamiento.

4. Modelos

En esta sección desarrollaremos algunos puntos en común y describiremos los modelos individuales que utilizamos, qué pruebas se hicieron en cada uno y los resultados que obtuvimos.

4.1 División de set de datos y métrica

En todos los casos usamos el mismo método de división (train_test_split de sklearn) con los valores default. Tuvimos especial cuidado en no contaminar el set de validación y siempre logramos localmente resultados de validación muy similares a los que luego obtuvimos en la competencia.

La métrica que utilizamos fue mean_absolute_error (también de sklearn), para mantener coherencia con la competencia. En cuanto al baseline, hay muchos que podríamos usar pero destacamos la solución por sample_zeros, con score aproximado 2.5M. De todas formas, como marco para evaluar nuestro puntaje utilizamos nuestros propios resultados anteriores y los del resto de los equipos que fueron subiendo a la competencia.

4.2 Modelo PromedioZona

Este fue el primer modelo que desarrollamos. No utiliza librerías externas y se basa en la hipótesis de que los factores más importantes a la hora de determinar el precio de una publicación son el tamaño de la propiedad y dónde está ubicada.

El modelo entrena armando un diccionario donde asigna a cada zona un valor por metro (por promedio). Luego, predice el precio multiplicando el tamaño de la propiedad por el valor por metro que tenga la zona a la que corresponde. Para hacer esto, trata de asignar la zona desde la más específica a la menos específica (idzona, ciudad, provincia) y en cada caso si encuentra un resultado lo devuelve. En caso de no tener algún dato, predice valores default.

A pesar de su simpleza, este modelo predice valores bastante acertados de precio, logrando un puntaje individual de alrededor de 850k puntos.

Suponiendo que encontraríamos mejores modelos, no se optimizó demasiado. Sin embargo, sí aportó positivamente en ensambles, ya que es muy rápido y aprovecha muy bien los pocos features que utiliza, considerando que estos features categóricos son muy costosos de usar en otros modelos.

4.3 Modelo RegresionLineal

Este fue el primer modelo que desarrollamos basado en una implementación externa (LinearRegression de sklearn). No lleva hiperparámetros y es extremadamente rápido para entrenar y predecir. En un principio los resultados eran mediocres (score 1.1M) pero al aplicar one hot encoding sobre los features tipodepropiedad y provincia logramos mejorarlo hasta llegar a un score de 870k. Individualmente es un puntaje bajo, pero en un ensamble puede sumar. Por otro lado, esta diferencia refuerza la hipótesis de que los features de locación son muy influyentes en el precio de la publicación.

4.4 Modelo XGBoostRegressor

Siendo uno de los algoritmos que mayor cantidad de competencias gana, decidimos probar qué resultados arroja en nuestro problema. El primer intento con sólo los features numéricos dio un resultado malo (970k) pero con al agregar one hot_encoding para tipodepropiedad y provincia logramos llegar a 770k y agregando también one hot encoding para ciudad alcanzamos los 738k, llegando hasta nuestro mejor puntaje hasta el momento y con un tiempo de entrenamiento no tan grande (menos de 10 minutos). Además, volvimos a comprobar que los features de locación son fundamentales.

Hasta este momento, habíamos venido usando los hiperparámetros default. Sabiendo que xgboost es muy sensible a los hiperparámetros, decidimos aplicar tuning para intentar mejorar el puntaje. Tras investigar los métodos de tuning proporcionados por la API de sklearn, nos dimos cuenta de que para poder usarlos iba a ser necesario repensar la forma en que estaba definida nuestra clase Modelo (y modificar los modelos que ya habíamos implementado). En consecuencia, decidimos hacer nuestro propio método de búsqueda de hiperparámetros.

El método es muy simple, lo alimentamos con un diccionario cuyas claves son los hiperparámetros a probar y los valores que pueden tomar. En base a esto genera algunos pocos casos de prueba aleatorios y busca el puntaje de validación en cada caso. Devuelve una lista con los grupos de hiperparámetros que probó y sus puntajes. Con los mejores casos, volvemos a armar el diccionario tratando de afinar la precisión y repetimos hasta que estemos conformes con el resultado. Es un poco engorroso porque requiere algo de supervisión, pero en muy poco tiempo lo tuvimos andando.

De esta forma, logramos encontrar hiperparámetros para xgboost que nos llevaron a mejorar el puntaje hasta 580k y subir unas cuantas posiciones en la tabla de la competencia.

4.5 Modelo XGBoostClassifier y predicción de faltantes

Para intentar mejorar los resultados obtenidos hasta el momento, decidimos analizar en qué estaban fallando nuestros modelos. De esta forma, dividimos las predicciones en dos grupos: las 100 publicaciones que obtuvieron mejores resultados y las 100 que obtuvieron peores resultados. No fue una sorpresa descubrir un patrón común que se repetía en las peores predicciones pero no en las mejores: datos faltantes.

Volviendo a nuestro set de datos original (combinando train, test y submit), contamos en cuántas publicaciones faltaba cada feature obteniendo la siguiente tabla:

Feature	Faltantes	Tipo de Dato
metrostotales	64122	numérico
antiguedad	54269	numérico
garages	47088	numérico / categórico
idzona	35800	categórico
banos	32775	numérico / categórico
habitaciones	28099	numérico / categórico
metroscubiertos	21699	numérico
ciudad	455	categórico
provincia	197	categórico
tipodepropiedad	53	categórico

Se nos ocurrió entonces tratar de predecir, para cada uno de estos features los valores faltantes. Hasta el momento sólo teníamos regresores, no clasificadores, por lo que implementamos un modelo de clasificador xgboost que recibe como parámetro qué feature predecir. El modelo además adapta los sets de datos para usar en train y test las publicaciones de las cuales se tienen datos y en submit las que se van a querer completar.

Hicimos algunas pruebas prediciendo algunos de los features categóricos faltantes pero no llegamos al 70% de efectividad en ningún caso y al usar los resultados no mejoró el puntaje. Esto puede explicarse porque el modelo más importante que teníamos hasta el momento era el xgboost, que por sí solo toma decisiones “inteligentes” respecto a los valores faltantes.

Viendo estos resultados y faltando todavía varios modelos para probar, decidimos abandonar este camino. Sin embargo, creemos que de aplicarse correctamente podría ayudar a mejorar resultados de algunos de los modelos.

4.6 Modelo MLP_Regressor

El último modelo individual que probamos fue una red MLP basada en la implementación de sklearn. Para poder usarlo tuvimos que aplicar una transformación de forma tal que los valores estén estandarizados. Este algoritmo también tiene hiperparámetros, por lo que volvimos a aplicar la metodología utilizada con xgboost regressor hasta encontrar los valores que mejor resultados nos dieron. El mejor puntaje individual que logramos obtener fue de 645k, decente pero lejos del xgboost. La gran desventaja en este caso es que el entrenamiento es mucho más demandante que los anteriores y demora más de una hora en cada pasada. Por este motivo, sólo probamos algunas de las combinaciones de hiperparámetros posibles quedando la duda de si podríamos mejorar el puntaje.

5. Ensambles

Una vez que tuvimos algunos modelos, empezamos a probar ensambles de los mismos abordando el problema desde dos ópticas distintas: por pesos y por concatenación.

5.1 Ensamble por pesos

En este caso, la estrategia fue asignar pesos a los distintos modelos y combinar las predicciones con un promedio ponderado según esos pesos. Hicimos algunas pruebas que mejoran los resultados individuales pero enseguida abandonamos viendo que la otra estrategia nos funcionaba mejor. Nos quedó pendiente buscar los pesos ideales como si fueran hiperparámetros de nuestro modelo ensamble. De esa forma quizás los resultados hubieran mejorado mucho.

5.2 Ensamble por concatenación - Mejor puntaje

Con esta estrategia de ensamble logramos el mejor puntaje del equipo. Hicimos distintas pruebas pero la base siempre fue la misma. Predecir con distintos modelos, agregar el resultado como un feature nuevo y luego alimentar al XGBoostRegressor con esos datos. El gran beneficio de esta estrategia es que el modelo final (xgboost en este caso) decide por sí mismo qué relevancia asignarle a las predicciones de los modelos de primer capa. Además, si bien no lo hicimos, con muy pocas mejoras podríamos tunear los hiperparámetros de la última capa para que sean óptimos respecto a la combinación previa de modelos.

Aplicando este ensamble llegamos al mejor puntaje: 530k, con tiempo de entrenamiento 1h30m. A continuación, un detalle de los tiempos y puntajes en este caso:

Capa	Modelo	Tiempo Entrenamiento	Tiempo Predicción	Validación
Inicial	RegresionLineal	4s	1s	869k
	PromedioZona	1s	7s	849k
	MLP_Regressor	4900s	4s	645k
Final	XGBoostRegressor	1800s	4s	533k

6. Conclusiones

A partir de los resultados obtenidos y basándonos en los resultados de otros equipos, confirmamos la hipótesis de que el problema de predicción de precios de publicaciones inmobiliarias se puede resolver con un nivel aceptable de error sin grandes requerimientos técnicos ni de tiempo.

Por otro lado, destacamos que el mismo problema puede abordarse desde una infinidad de modelos y que en cada caso las variantes son muchas también. En este trabajo sólo se probaron algunas de esas posibilidades pero ello no quiere decir que estas sean las mejores para este problema necesariamente. Asimismo, existe también un margen de mejora con el tuning de hiperparámetros con la elaboración de nuevos features y con la limpieza profunda del set de datos.