024-price-and-everything

April 29, 2022

Predicting Price with Size, Location, and Neighborhood

[101]:
```
import warnings
from glob import glob

import pandas as pd
import seaborn as sns
import wqet_grader
from category_encoders import OneHotEncoder
from IPython.display import VimeoVideo
from ipywidgets import Dropdown, FloatSlider, IntSlider, interact
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression, Ridge  # noqa F401
from sklearn.metrics import mean_absolute_error
from sklearn.pipeline import make_pipeline
from sklearn.utils.validation import check_is_fitted

warnings.simplefilter(action="ignore", category=FutureWarning)
wqet_grader.init("Project 2 Assessment")
```

```
<IPython.core.display.HTML object>
```

In the final lesson for this project, we're going to try to use all the features in our dataset to improve our model. This means that we'll have to do a more careful cleaning of the dataset and consider some of the finer points of linear models.

[102]:
```
VimeoVideo("656842813", h="07f074324e", width=600)
```

[102]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e6059040>
```

1 Prepare Data

1.1 Import

[103]:
```
def wrangle(filepath):
    # Read CSV file
    df = pd.read_csv(filepath)
```

```
    # Subset data: Apartments in "Capital Federal", less than 400,000
    mask_ba = df["place_with_parent_names"].str.contains("Capital Federal")
    mask_apt = df["property_type"] == "apartment"
    mask_price = df["price_aprox_usd"] < 400_000
    df = df[mask_ba & mask_apt & mask_price]

    # Subset data: Remove outliers for "surface_covered_in_m2"
    low, high = df["surface_covered_in_m2"].quantile([0.1, 0.9])
    mask_area = df["surface_covered_in_m2"].between(low, high)
    df = df[mask_area]

    # Split "lat-lon" column
    df[["lat", "lon"]] = df["lat-lon"].str.split(",", expand=True).astype(float)
    df.drop(columns="lat-lon", inplace=True)

    # Get place name
    df["neighborhood"] = df["place_with_parent_names"].str.split("|",␣
↪expand=True)[3]
    df.drop(columns="place_with_parent_names", inplace=True)

    df.drop(columns = ['floor', 'expenses'], inplace = True)

    df.drop(columns = ['operation', 'property_type', 'currency',␣
↪'properati_url'], inplace = True)

    df.drop(columns = ['price',
                       'price_aprox_local_currency',
                       'price_per_m2',
                       'price_usd_per_m2' ], inplace = True)

    df.drop(columns = ['surface_total_in_m2', 'rooms'], inplace = True)



    return df
```

Let's begin by using what we've learned to load all our CSV files into a DataFrame.

[104]:
```
VimeoVideo("656842538", h="bd85634eb1", width=600)
```

[104]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e62d6490>
```

Task 2.4.1: Use `glob` to create a list that contains the filenames for all the Buenos Aires real estate CSV files in the `data` directory. Assign this list to the variable name `files`.

- Assemble a list of path names that match a pattern in glob.

[105]:
```
files = sorted(glob('data/buenos-aires-real-estate-*.csv'))
files
```

[105]:
```
['data/buenos-aires-real-estate-1.csv',
 'data/buenos-aires-real-estate-2.csv',
 'data/buenos-aires-real-estate-3.csv',
 'data/buenos-aires-real-estate-4.csv',
 'data/buenos-aires-real-estate-5.csv']
```

[106]:
```
# Check your work
assert len(files) == 5, f"`files` should contain 5 items, not {len(files)}"
```

The last time we put all our DataFrames into a list, we used a `for` loop. This time, we're going to use a more compact coding technique called a **list comprehension**.

[107]:
```
VimeoVideo("656842076", h="0f654d427f", width=600)
```

[107]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e62d6ac0>
```

Task 2.4.2: Use your `wrangle` function in a list comprehension to create a list named `frames`. The list should contain the cleaned DataFrames for the filenames your collected in `files`.

- What's a list comprehension?
- Write a list comprehension in Python

[108]:
```
frames = [wrangle(file) for file in files]
```

[109]:
```
# Check your work
assert len(frames) == 5, f"`frames` should contain 5 items, not {len(frames)}"
assert all(
    [isinstance(frame, pd.DataFrame) for frame in frames]
), "The items in `frames` should all be DataFrames."
```

Last step: Combine the DataFrames in `frames` into a single `df`.

[110]:
```
VimeoVideo("656841910", h="79c7dbc5ab", width=600)
```

[110]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e63344c0>
```

Task 2.4.3: Use `pd.concat` to concatenate it items in `frames` into a single DataFrame `df`. Make sure you set the `ignore_index` argument to `True`.

- Concatenate two or more DataFrames using pandas.

[111]:
```
df = pd.concat(frames, ignore_index = True)
print(df.info())
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6582 entries, 0 to 6581
```

```
Data columns (total 5 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   price_aprox_usd        6582 non-null   float64
 1   surface_covered_in_m2  6582 non-null   float64
 2   lat                    6316 non-null   float64
 3   lon                    6316 non-null   float64
 4   neighborhood           6582 non-null   object
dtypes: float64(4), object(1)
memory usage: 257.2+ KB
None
```

[111]:

	price_aprox_usd	surface_covered_in_m2	lat	lon	neighborhood
0	129000.0	70.0	-34.584651	-58.454693	Chacarita
1	87000.0	42.0	-34.638979	-58.500115	Villa Luro
2	118000.0	54.0	-34.615847	-58.459957	Caballito
3	57000.0	42.0	-34.625222	-58.382382	Constitución
4	90000.0	50.0	-34.610610	-58.412511	Once

[112]:
```
# Check your work
assert len(df) == 6582, f"`df` has the wrong number of rows: {len(df)}"
assert df.shape[1] <= 17, f"`df` has too many columns: {df.shape[1]}"
```

1.2 Explore

The first thing we need to consider when trying to use all the features `df` is **missing values**. While it's true you can impute missing values, there still needs to be enough data in a column to do a good imputation. A general rule is that, if more than half of the data in a column is missing, it's better to drop it then try imputing.

Take a look at the output from `df.info()` above. Are there columns where more than half of the values are `NaN`? If so, those columns need to go!

[113]:
```
VimeoVideo("656848648", h="6964fa0c8c", width=600)
```

[113]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e6334520>
```

Task 2.4.4: Modify your `wrangle` function to drop any columns that are more than half `NaN` values. Be sure to rerun all the cells above before you continue.

- Inspect a DataFrame using the `shape`, `info`, and `head` in pandas.
- Drop a column from a DataFrame using pandas.

[114]:
```
# Check your work
assert len(df) == 6582, f"`df` has the wrong number of rows: {len(df)}"
assert df.shape[1] <= 15, f"`df` has too many columns: {df.shape[1]}"
```

The next thing we need to look out for are categorical columns with **low or high cardinality**. If there's only one category in a column, it won't provide any unique information to our model. At

the other extreme, columns where nearly every row has its own category won't help our model in identifying useful trends in the data.

Let's take a look at the cardinality of our features.

[115]:
```
VimeoVideo("656848196", h="37dbc44b09", width=600)
```

[115]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e6334610>
```

Task 2.4.5: Calculate the number of unique values for each non-numeric feature in `df`.

- Subset a DataFrame's columns based on the column data types in pandas.
- Calculate summary statistics for a DataFrame or Series in pandas.

[116]:
```
df.select_dtypes('object').head()
```

[116]:
```
   neighborhood
0     Chacarita
1    Villa Luro
2     Caballito
3  Constitución
4          Once
```

[117]:
```
df.select_dtypes('object').nunique()
```

[117]:
```
neighborhood    57
dtype: int64
```

Here, we can see that columns like `"operation"` have only one value in them, while every row in `"properati_url"` has a unique value. These are clear examples of high- and low-cardinality features that we shouldn't include in our model.

Task 2.4.6: Modify your `wrangle` function to drop high- and low-cardinality categorical features.

Be sure to rerun all the cells above before you continue.

- What are high- and low-cardinality features?
- Drop a column from a DataFrame using pandas.

[118]:
```
# Check your work
assert len(df) == 6582, f"`df` has the wrong number of rows: {len(df)}"
assert df.shape[1] <= 11, f"`df` has too many columns: {df.shape[1]}"
```

It's also important for us to drop any columns that would constitute **leakage**, that is, features that were created using our target or that would give our model information that it won't have access to when it's deployed.

[119]:
```
VimeoVideo("656847896", h="11de775937", width=600)
```

[119]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e1f2ed00>
```

Task 2.4.7: Modify your `wrangle` function to drop any features that would constitute leakage.

Be sure to rerun all the cells above before you continue.

- What's leakage?
- Drop a column from a DataFrame using pandas.

[120]:
```
sorted(df.columns)
```

[120]:
```
['lat', 'lon', 'neighborhood', 'price_aprox_usd', 'surface_covered_in_m2']
```

[121]:
```
# Check your work
assert len(df) == 6582, f"`df` has the wrong number of rows: {len(df)}"
assert df.shape[1] <= 7, f"`df` has too many columns: {df.shape[1]}"
```

Finally, the last issue we need to keep an eye out for is **multicollinearity**, that is, features in our feature matrix that are highly correlated with each other. A good way to detect this is to use a heatmap. Let's make one!

[81]:
```
VimeoVideo("656847237", h="4b5cfed5d6", width=600)
```

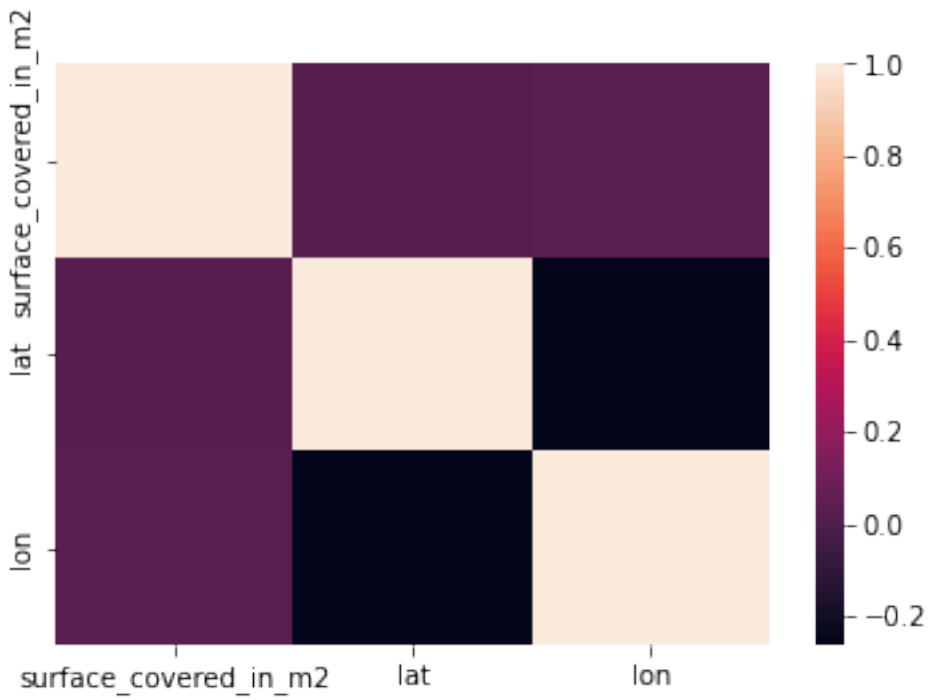
[81]:
```
<IPython.lib.display.VimeoVideo at 0x7fd200f66490>
```

Task 2.4.8: Plot a correlation heatmap of the remaining numerical features in `df`. Since `"price_aprox_usd"` will be your target, you don't need to include it in your heatmap.

- What's a heatmap?
- Create a correlation matrix in pandas.
- Create a heatmap in seaborn.

[122]:
```
corr = df.select_dtypes('number').drop(columns = 'price_aprox_usd').corr()
sns.heatmap(corr)
```

[122]:
```
<AxesSubplot:>
```



[123]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6582 entries, 0 to 6581
Data columns (total 5 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   price_aprox_usd        6582 non-null   float64
 1   surface_covered_in_m2  6582 non-null   float64
 2   lat                    6316 non-null   float64
 3   lon                    6316 non-null   float64
 4   neighborhood           6582 non-null   object
dtypes: float64(4), object(1)
memory usage: 257.2+ KB
```

Task 2.4.9: Modify your `wrangle` function to remove columns so that there are no strongly correlated features in your feature matrix.

Be sure to rerun all the cells above before you continue.

- What's multicollinearity?
- Drop a column from a DataFrame using pandas.

[124]:
```
# Check your work
assert len(df) == 6582, f"`df` has the wrong number of rows: {len(df)}"
assert df.shape[1] == 5, f"`df` has the wrong number of columns: {df.shape[1]}"
```

```
df.head()
```

[124]:

	price_aprox_usd	surface_covered_in_m2	lat	lon	neighborhood
0	129000.0	70.0	-34.584651	-58.454693	Chacarita
1	87000.0	42.0	-34.638979	-58.500115	Villa Luro
2	118000.0	54.0	-34.615847	-58.459957	Caballito
3	57000.0	42.0	-34.625222	-58.382382	Constitución
4	90000.0	50.0	-34.610610	-58.412511	Once

Done! It looks like we're going to use the four features we've used in our previous models but, this time, we're going to combine them.

1.3 Split Data

Task 2.4.10: Create your feature matrix `X_train` and target vector `y_train`. Your target is `"price_aprox_usd"`. Your features should be all the columns that remain in the DataFrame you cleaned above.

- What's a feature matrix?
- What's a target vector?
- Subset a DataFrame by selecting one or more columns in pandas.
- Select a Series from a DataFrame in pandas.

[125]:
```
target = "price_aprox_usd"
features = df.drop(columns = 'price_aprox_usd').columns

X_train = df[features]
y_train = df[target]
```

[126]:
```
# Check your work
assert X_train.shape == (6582, 4), f"`X_train` is the wrong size: {X_train.
 ↪shape}."
assert y_train.shape == (6582,), f"`y_train` is the wrong size: {y_train.shape}.
 ↪"
```

2 Build Model

2.1 Baseline

[127]:
```
VimeoVideo("656849559", h="bca444c8af", width=600)
```

[127]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e6059070>
```

Task 2.4.11: Calculate the baseline mean absolute error for your model.

- Calculate summary statistics for a DataFrame or Series in pandas.

[128]:
```
y_mean = y_train.mean()
y_pred_baseline = [y_mean] * len(y_train)

print("Mean apt price:", round(y_mean, 2))

print("Baseline MAE:", mean_absolute_error(y_train, y_pred_baseline))
```

```
Mean apt price: 132383.84
Baseline MAE: 44860.10834274134
```

2.2 Iterate

Task 2.4.12: Create a pipeline named `model` that contains a `OneHotEncoder`, `SimpleImputer`, and `Ridge` predictor.

- What's imputation?
- What's one-hot encoding?
- What's a pipeline?
- Create a pipeline in scikit-learn.

[130]:
```
model = make_pipeline(
    OneHotEncoder(use_cat_names = True),
    SimpleImputer(),
    Ridge()
)

model.fit(X_train, y_train)
```

[130]:
```
Pipeline(steps=[('onehotencoder',
                 OneHotEncoder(cols=['neighborhood'], use_cat_names=True)),
                ('simpleimputer', SimpleImputer()), ('ridge', Ridge())])
```

[131]:
```
# Check your work
check_is_fitted(model[-1])
```

2.3 Evaluate

[132]:
```
VimeoVideo("656849505", h="f153a4f005", width=600)
```

[132]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e1e1c160>
```

Task 2.4.13: Calculate the training mean absolute error for your predictions as compared to the true targets in `y_train`.

- Generate predictions using a trained model in scikit-learn.
- Calculate the mean absolute error for a list of predictions in scikit-learn.

[133]:
```
y_pred_training = model.predict(X_train)
print("Training MAE:", mean_absolute_error(y_train, y_pred_training))
```

```
Training MAE: 24207.10719033026
```

Task 2.4.14: Run the code below to import your test data `buenos-aires-test-features.csv` into a DataFrame and generate a list of predictions using your model. Then run the following cell to submit your predictions to the grader.

- What's generalizability?
- Generate predictions using a trained model in scikit-learn.
- Calculate the mean absolute error for a list of predictions in scikit-learn.

[134]:
```
X_test = pd.read_csv("data/buenos-aires-test-features.csv")
y_pred_test = pd.Series(model.predict(X_test))
y_pred_test.head()
```

[134]:
```
0    231122.403569
1    162572.942392
2     68477.949626
3     63521.438989
4    105694.463885
dtype: float64
```

[135]:
```
wqet_grader.grade("Project 2 Assessment", "Task 2.4.14", y_pred_test)
```

```
<IPython.core.display.HTML object>
```

3 Communicate Results

For this lesson, we've relied on equations and visualizations for communication about our model. In many data science projects, however, communication means giving stakeholders tools they can use to **deploy** a model — in other words, use it in action. So let's look at two ways you might deploy this model.

One thing you might be asked to do it wrap your model in a function so that a programmer can provide inputs and then receive a prediction as output.

[136]:
```
VimeoVideo("656849254", h="e6faad47ca", width=600)
```

[136]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e1e16670>
```

Task 2.4.15: Create a function `make_prediction` that takes four arguments (`area`, `lat`, `lon`, and `neighborhood`) and returns your model's prediction for an apartment price.

[143]:
```
def make_prediction(area, lat, lon, neighborhood):
    data = {
        'surface_covered_in_m2': area,
        'lat': lat,
        'lon': lon,
        'neighborhood': neighborhood
    }
```

```
    df = pd.DataFrame(data, index = [0])
    prediction = model.predict(df).round(2)[0]
    return f"Predicted apartment price: ${prediction}"
```

Let's see if your function works. Run the cell below to find out!

[144]:
```
make_prediction(110, -34.60, -58.46, "Villa Crespo")
```

[144]:
```
'Predicted apartment price: $250775.11'
```

Another type of deployment is creating and interactive dashboard, where a user can supply values and receive a prediction. Let's create one using Jupyter Widgets.

[145]:
```
VimeoVideo("656848911", h="7939dcd479", width=600)
```

[145]:
```
<IPython.lib.display.VimeoVideo at 0x7fd1e1e1e940>
```

Task 2.4.16: Add your `make_prediction` to the interact widget below, run the cell, and then adjust the widget to see how predicted apartment price changes.

- Create an interact function in Jupyter Widgets.

[146]:
```
interact(
    make_prediction,
    area=IntSlider(
        min=X_train["surface_covered_in_m2"].min(),
        max=X_train["surface_covered_in_m2"].max(),
        value=X_train["surface_covered_in_m2"].mean(),
    ),
    lat=FloatSlider(
        min=X_train["lat"].min(),
        max=X_train["lat"].max(),
        step=0.01,
        value=X_train["lat"].mean(),
    ),
    lon=FloatSlider(
        min=X_train["lon"].min(),
        max=X_train["lon"].max(),
        step=0.01,
        value=X_train["lon"].mean(),
    ),
    neighborhood=Dropdown(options=sorted(X_train["neighborhood"].unique())),
);
```

```
interactive(children=(IntSlider(value=53, description='area', max=101, min=30),␣
↪FloatSlider(value=-34.59890626…
```

Great work! You may have noticed that there a lots of ways to improve this dashboard. For instance, a user can select a neighborhood and then supply latitude-longitude coordinates that aren't in that

neighborhood. It would also be helpful to include a visualization like a map. Regardless, this is a great first step towards creating dynamic dashboards that turn your model from a complicated abstraction to a concrete tool that anyone can access. One of the most important parts of data science projects is creating products that people can use to make their work or lives easier.
