Data Science Lab

Lab 9 solution

Exercise 1

The goal of this exercise is to predict, based on the available data of Airbnb posts, the requested renting price. Let us begin by loading the datasets into memory. We can use pandas' `read_csv()` method to obtain ready-to-use DataFrames.

[1]:
```
import pandas as pd

# we are using the "id" column as the effective id (column 0) for each row
df_dev = pd.read_csv("NYC_Airbnb/development.csv", index_col=0)
df_eval = pd.read_csv("NYC_Airbnb/evaluation.csv", index_col=0)
```

[2]:
```
df_dev.head()
```

[2]:
```
                                                  name   host_id  \
id
12783632                                 NYC Mini Hotel  57230304
3463385                        Gorgeous room in Manhattan  10698270
17572926                    Great 1 Bedroom on Upper East  36578169
33913644  Modern and bright 2Bed 2Bath Bushwick, Brooklyn  50981314
9405895                   Stylish and zen Brooklyn retreat  48775347

          host_name neighbourhood_group    neighbourhood  latitude  longitude  \
id
12783632  Imanuelly              Queens         Elmhurst  40.74037  -73.88610
3463385     Evgenia           Manhattan  Upper East Side  40.76717  -73.95532
17572926      James           Manhattan  Upper East Side  40.77984  -73.94725
33913644      Ofier            Brooklyn         Bushwick  40.70205  -73.91338
9405895     Mathieu            Brooklyn      Fort Greene  40.68914  -73.97853

                room_type  price  minimum_nights  number_of_reviews  \
id
12783632     Private room     75               1                  2
3463385      Private room     95               1                202
17572926  Entire home/apt    130               2                  0
33913644  Entire home/apt    150               2                  4
9405895   Entire home/apt    325               3                 16
```

```
         last_review  reviews_per_month  calculated_host_listings_count  \
id
12783632  2019-05-26               0.92                               3
3463385   2019-05-27               3.31                               2
17572926         NaN                NaN                               1
33913644  2019-07-07               1.64                               1
9405895   2019-04-20               0.42                               1

          availability_365
id
12783632               351
3463385                263
17572926                 0
33913644                89
9405895                103
```

We know, from the problem description, that the development and the evaluation sets only differ for one column, `price` (our target column, which is obviously not included in the evaluation file).

We can take a first look at the columns that comprise our data and make some initial considerations based on it.

[3]:
```
df_dev.columns
```

[3]:
```
Index(['name', 'host_id', 'host_name', 'neighbourhood_group', 'neighbourhood',
       'latitude', 'longitude', 'room_type', 'price', 'minimum_nights',
       'number_of_reviews', 'last_review', 'reviews_per_month',
       'calculated_host_listings_count', 'availability_365'],
      dtype='object')
```

0. `id`: this is an incremental id that uniquely identifies each posting. The ids may not be contiguous as they only cover places in NYC. This is not supposed to be a predictive variable but, as we will see later on, some useful information may have "leaked" in this variable
1. `name`: this represents the title of the posting. This title is in natural language and can be processed with text mining techniques already shown in this course. Intuitively, this column might contain useful information. For example, the text "1/2/3/n bedrooms" might give away additional information on the actual size of the condo, "wall street", "central park", "soho" and other words might provide additional information on the exact location of the place, "amazing", "new", "recent" might give away information on the conditions of the place
2. `host_id`: this provides information on the id of the user who posted the ad (the "host"). Once again, this is an incremental id and, at a first glance, it may not contain any useful information. However, single accounts may have posted multiple ads for multiple places. This is something we could actually learn from as, intuitively, it is likely that the same person/company owns places with similar price ranges. As such, we may want to look into the number of hosts with more than one ad published. Additionally, as it happens with `id`, there might be some useful information that has "leaked" into `host_id`. We will see more about this later

3. host_name: this is expected to have a 1:1 mapping with host_id, as it represents the username of the user who posted the ad. However, since many of the users may not have set the username, or have asked Airbnb not to include it, this 1:1 mapping does not actually hold. Intuitively, this column should not introduce any additional information that is not already present in host_id
4. neighbourhood_group: this is a string that identifies the neighbourhood the place is in. By extracting a set of this column, we get these five neighbourhood groups: Bronx, Brooklyn, Manhattan, Queens, Staten Island. The location could clearly help define how expensive a place can be (different neighbourhoods clearly have different prices)
5. neighbourhood: this provides a more fine-grained information than neighbourhood_group. It divides the neighbourhood groups into smaller "chunks", for a total of 221 neighbourhoods. This could help better define where a place is and, therefore, how expensive it can be
6. latitude, longitude: these are the actual coordinates of the apartment. This provides the most fine-grained kind of information possible. This is the "raw" location and has not already been preprocessed to a higher, more abstract, level (such as that of neighbourhoods). This could either be a good thing (if neighbourhoods do not clearly define price ranges) or a bad thing (if the already avaialble neighbourhood are good indicators, the model we train will not benefit from the latitude/longitude information)
7. room_type: this represents the kind of place that is being offerend. By extracting a set out of this column, we get these three values: "Entire home/apt", "Private room", "Shared room". These values are quite self-explanatory and will not be discussed further. It is, though, quite clear that they should be an important price driver
8. price: this is the target column, the price. It is only available for df_dev (and not for df_eval)
9. minimum_nights: this represents the minimum number of nights that customers are required to stay to make a reservation. It might, in some way, be affected by the price (for example it might be that more luxurious places should typically be booked for a larger number of nights (e.g. to offset some fixed expenses))
10. number_of_reviews: this is the total number of reviews received for a posting. At a first glance, it might not be particularly meaningful for predicting the requested price, but one may argue that high-end places are typically booked by a lower number of customers. This would in turn result in a lower number of overall reviews
11. last_review: this is the date when the latest review was left for a listing. It might be useful as it may contain information similar to that of number_of_reviews. However, precisely because it is likely to include the same information as number_of_reviews, it might not introduce any useful information (and, if we were to discard it, we would not have to deal with dates)
12. reviews_per_month: as with last_review, this might be useful to determine the price of a listing. Additionally, since this is a "per month" kind of information, this column is not affected by how long the listing has been published.
13. calculated_host_listings_count: this column defines the number of listings posted by the host
14. availability_365: this column defines the number of days per year where this listing is available. One may argue that very exclusive locations might only be available at certain times of the year.

As you can see, almost all features might be argued to have some sort of relationship with the target value. Since we only have a limited number of features, we can reasonably keep all of the features we suspect might have some kind of relationship with the price. It will be our regressor to understand whether those features are useful or not for our problem.

We can now proceed to some preliminary analyses. For these, we will use a merged version of `df_dev` and `df_eval`, which we will call `df`. In this way, we can run our code only once on the entire dataset. Clearly, when studying `df`, we will not consider the `price` column (since that column is only available on `df_dev`).

[4]:
```
df = pd.concat([df_dev, df_eval], sort=False)
len(df_dev), len(df_eval), len(df)
```

[4]:
```
(39116, 9779, 48895)
```

A potentially interesting information regards the existance of empty (or Not a Number) values.

[5]:
```
df.isna().any(axis=0)
```

[5]:
```
name                               True
host_id                           False
host_name                          True
neighbourhood_group               False
neighbourhood                     False
latitude                          False
longitude                         False
room_type                         False
price                              True
minimum_nights                    False
number_of_reviews                 False
last_review                        True
reviews_per_month                  True
calculated_host_listings_count    False
availability_365                  False
dtype: bool
```

We can see that `name` might contain empty values. While surprising (we would expect each posting to be assigned a title), it should not be particularly hard to handle missing values: an empty title should suffice (considering that there is no way for us to infer this value elsewhere).

As already mentioned, `host_name` might be empty: this could be because the user decided not to have it shown. Since we already know that `host_id` contains all the useful information we could get from `host_name`, we could drop this column without worrying about handling missing values.

`price`, surprisingly, appears to be having missing values. Remember, though, that `df` is the merged version of `df_dev` and `df_eval`. We built this DataFrame using `pd.concat`. By default, this function will fill any mismatching columns (i.e. columns that exist in one DataFrame but not the other) with NaN values. This is exactly what happened here. Indeed, if we check `df_dev` only:

[6]:
```
df_dev["price"].isna().any()
```

[6]: False

we discover that df_dev has all the price values.

Finally, we find out that both last_review and reviews_per_month can be NaN. A reasonable explanation could be that, for those listings that have received no reviews (number_of_reviews = 0) there is no way of defining those other two values. We can verify this assertion rather simply:

[7]:
```
(df[df["number_of_reviews"]==0].index == df[df["last_review"].isna()].index).
 ↪all(),\
(df[df["number_of_reviews"]==0].index == df[df["reviews_per_month"].isna()].
 ↪index).all()
```

[7]: (True, True)

The first part tells us that all the rows that have received no reviews (df[df["number_of_reviews"]==0]) have the same list of indices as the rows that have a NaN last_review. The second part contains the same information, for reviews_per_month. As already stated, we can decide to ignore the last_review feature, and we can initialize reviews_per_month to 0 for those listings that have not received any review.

[8]:
```
df["reviews_per_month"].fillna(0, inplace=True) # with inplace, we make the␣
 ↪changes directly to df
```

Now, for the next part, we need to convert any non-numerical (i.e. categorical) feature into a numerical one. This is because machine learning models work on numeric data and cannot digest non-numeric data without some kind of transformation.

From our previous assessment we have identified three categorical features that need to be transformed into numerical ones: neighbourhood_group, neighbourhood, room_type.

We should be careful, though, when performing a conversion of a categorical variable into a numerical one. A naive approach would be to assign a number to each possible value of the column (for neighbourhood_group, for example, we could use Bronx: 0, Brooklyn: 1, Manhattan: 2, Queens: 3, Staten Island: 4). If we do this, though, we are introducing an order among the values that did not exist before: we are saying, for example, that Queens is "larger than" Brooklyn but "smaller than" Staten Island. This clearly does not make any sense: since machine learning models could pick up on these orders (think about how decision trees make their splits) we should therefore discard this option, in favor of a different one.

An approach that is typically used is the 1-hot encoding. A column that can have N distinct values is converted into N boolean columns: for each row, only one of the N columns will be set to 1 (i.e. the column associated with the value of that row for the original column), the others will be 0's.

Pandas offer a function for converting categorical features into 1-hot encoded columns: pd.get_dummies.

[9]:
```
df_1h = pd.get_dummies(df, columns=['neighbourhood_group', 'neighbourhood',␣
 ↪'room_type'])
```

[10]:
```
df.shape, df_1h.shape
```

[10]:
```
((48895, 15), (48895, 241))
```

By applying the 1-hot encoding, we are converting our original 15 columns into 241. This is because we have 221 possible values for `neighbourhood`, 5 for `neighbourhood_group` and 3 for `room_type`, but we are removing the three original columns:

[11]:
```
15 + 221 + 5 + 3 - 3
```

[11]:
```
241
```

We now have enough information to build an initial regressor. Based on the algorithm we decide to use, we may or may not need to do some further normalization. It makes sense to use a random forest regressor, given how well it typically performs and given its interpretability (we can extract the importance of each feature from it, to understand whether our initial assumptions were reasonable).

Since random forests are based on decision trees and since decision trees work on one feature at a time, there is no need to normalize the dataset just yet. We should, though, build a train/validation and test sets.

[12]:
```
from sklearn.model_selection import train_test_split

# drop unused columns
df_dropped = df_1h.drop(columns=["host_id", "name", "host_name", "last_review"])
# define the mask for the training/validation samples (those with a price, the␣
↪others
# will belong to the test set)
train_valid_mask = ~df_dropped["price"].isna()
# extract the feature names (for later use)
feature_names = df_dropped[train_valid_mask].drop(columns=["price"]).columns

X = df_dropped.drop(columns=["price"]).values
y = df_dropped["price"].values

X_train_valid = X[train_valid_mask]
y_train_valid = y[train_valid_mask]
X_test = X[~train_valid_mask]
y_test = y[~train_valid_mask]

X_train, X_valid, y_train, y_valid = train_test_split(X_train_valid,␣
↪y_train_valid, shuffle=True, random_state=42)
```

First, we discard the features that we are currently not considering. These include `id` (which is automatically discarded when exporting the NumPy array (`.values`), `host_id` and `host_name` (which we have currently deemed as not particularly helpful), `name` (which definitely contains useful information and we may consider reintroducing later on) and `last_review` (which, as already stated, contains information already available in other columns of the datasets).

We can now train a random forest with the default parameters to establish an initial baseline.

[13]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score

reg = RandomForestRegressor(100, random_state=42)
reg.fit(X_train , y_train)
r2_score(y_valid, reg.predict(X_valid))
```

[13]:
```
0.07128877883880957
```

This is the baseline value we get on our validation data.

The random foret regressor we have trained can help us determine how useful each feature is for the model. We can extract this feature importance and sort them in descending order. This will let us know where to focus next.

[14]:
```
sorted(zip(feature_names, reg.feature_importances_), key=lambda x: x[1],␣
↪reverse=True)
```

[14]:
```
[('longitude', 0.20115326835304928),
 ('latitude', 0.20071189574910814),
 ('availability_365', 0.09798847469596444),
 ('minimum_nights', 0.08811410026910035),
 ('reviews_per_month', 0.06634125348301373),
 ('room_type_Entire home/apt', 0.06307473812272034),
 ('number_of_reviews', 0.048997123358966936),
 ('calculated_host_listings_count', 0.04604768765360843),
 ('neighbourhood_Astoria', 0.03845108649925162),
 ('neighbourhood_Battery Park City', 0.020259916690909884),
 ('neighbourhood_Upper West Side', 0.012948624370908726),
 ('neighbourhood_Lower East Side', 0.012170110438514237),
 ('neighbourhood_Bedford-Stuyvesant', 0.009808512052124175),
 ('neighbourhood_Clinton Hill', 0.008191871889363506),
 ('neighbourhood_Randall Manor', 0.006261612568861734),
 ('neighbourhood_Tribeca', 0.00437265410943937),
 ('neighbourhood_Chelsea', 0.004251043171635635),
 ('neighbourhood_Midtown', 0.003967140823498989),
 ('neighbourhood_Williamsburg', 0.0038247610837308125),
 ('neighbourhood_Bay Ridge', 0.003762925586806024),
 ('neighbourhood_West Village', 0.003487549745255586),
 ('neighbourhood_Gramercy', 0.003148555182695803),
 ('neighbourhood_Greenwich Village', 0.002997310033222382 3),
 ('neighbourhood_Theater District', 0.002985703222628324 8),
 ('neighbourhood_East Village', 0.0028787241661582413),
 ('neighbourhood_group_Manhattan', 0.002843257309550643),
 ('neighbourhood_Riverdale', 0.002451341735075149),
 ('neighbourhood_Flatiron District', 0.0024063897298585504),
```

```
('neighbourhood_SoHo', 0.002399031003529347),
('neighbourhood_Upper East Side', 0.0021735264549634426),
('neighbourhood_East Harlem', 0.0019818163544116658),
('neighbourhood_Prospect-Lefferts Gardens', 0.001956568669750475),
('neighbourhood_group_Queens', 0.001914211514102117),
('neighbourhood_Prospect Heights', 0.0016929615589808416),
('neighbourhood_Forest Hills', 0.00152837188481164),
('neighbourhood_Murray Hill', 0.0012262088824587644),
('neighbourhood_Financial District', 0.00119426826792937),
('neighbourhood_Cypress Hills', 0.001157620936268546),
('neighbourhood_group_Brooklyn', 0.0010294616294437623),
('neighbourhood_Sunset Park', 0.0010139779338424167),
('neighbourhood_Long Island City', 0.0009815050537152383),
("neighbourhood_Hell's Kitchen", 0.0009688928162883463),
('neighbourhood_Sea Gate', 0.0007657577228048119),
('neighbourhood_Flatlands', 0.0007627954438911617),
('neighbourhood_Brighton Beach', 0.0007340222418291512),
('neighbourhood_Kips Bay', 0.0007248846378943465),
('neighbourhood_Crown Heights', 0.0007107578620415585),
('neighbourhood_NoHo', 0.0007028361378912315),
('neighbourhood_Stuyvesant Town', 0.000610225156226864),
('neighbourhood_Greenpoint', 0.0006095885120862495),
('neighbourhood_Cobble Hill', 0.0005692353517283695),
('neighbourhood_Chinatown', 0.0005291052055840536),
('neighbourhood_Bushwick', 0.0004924227001733878),
('neighbourhood_Roosevelt Island', 0.0004087742662780565),
('neighbourhood_East Flatbush', 0.0003964934258953782),
('room_type_Shared room', 0.0003869395936027991),
('neighbourhood_Park Slope', 0.0003806591572802716),
('room_type_Private room', 0.0003377518246210594),
('neighbourhood_Arverne', 0.000312087129763104),
('neighbourhood_Boerum Hill', 0.0002993795227682120 3),
('neighbourhood_Flatbush', 0.0002790627971694921),
('neighbourhood_Briarwood', 0.0002718382598977949),
('neighbourhood_Brooklyn Heights', 0.0002712404734392449 6),
('neighbourhood_Jamaica', 0.0002676278336615392),
('neighbourhood_Fort Greene', 0.0002657135899571233),
("neighbourhood_Prince's Bay", 0.000238674593580325),
('neighbourhood_Gowanus', 0.00023450889624785677),
('neighbourhood_Harlem', 0.0002303051846783579),
('neighbourhood_Little Italy', 0.00022137715110362132),
('neighbourhood_Nolita', 0.00019735068772511015),
('neighbourhood_Far Rockaway', 0.00015074620804473324),
('neighbourhood_Canarsie', 0.00012563872987656705),
('neighbourhood_Civic Center', 0.00011262729490357799),
('neighbourhood_Westchester Square', 0.00010959618583996965),
('neighbourhood_Carroll Gardens', 9.20735718387971e-05),
```

```
('neighbourhood_Elmhurst', 9.034931259534385e-05),
('neighbourhood_Woodside', 8.780890784466599e-05),
('neighbourhood_Washington Heights', 8.085649767634438e-05),
('neighbourhood_Mott Haven', 6.83708186689811e-05),
('neighbourhood_Morningside Heights', 6.60897890601337e-05),
('neighbourhood_South Slope', 5.9409687681036586e-05),
('neighbourhood_Vinegar Hill', 5.847705700304092e-05),
('neighbourhood_Arrochar', 5.634610423320641e-05),
('neighbourhood_Windsor Terrace', 5.606191701075737e-05),
('neighbourhood_St. Albans', 5.585431778955482e-05),
('neighbourhood_Clason Point', 5.1748531947672034e-05),
('neighbourhood_Pelham Gardens', 4.9105621477612914e-05),
('neighbourhood_Jamaica Estates', 4.58410053634476e-05),
('neighbourhood_Ditmars Steinway', 4.045757485618073e-05),
('neighbourhood_Sheepshead Bay', 3.888171592697607e-05),
('neighbourhood_Todt Hill', 3.7323402514148614e-05),
('neighbourhood_Flushing', 3.514057843959227e-05),
('neighbourhood_Downtown Brooklyn', 3.30377906586379e-05),
('neighbourhood_group_Bronx', 3.095365185458409e-05),
('neighbourhood_Rego Park', 3.052520728489699e-05),
('neighbourhood_Spuyten Duyvil', 2.9208118238816133e-05),
('neighbourhood_Kensington', 2.9138756433463438e-05),
('neighbourhood_Allerton', 2.88946942490595e-05),
('neighbourhood_Jackson Heights', 2.8360662693093722e-05),
('neighbourhood_DUMBO', 2.8157135523454246e-05),
('neighbourhood_Richmond Hill', 2.695501195200147e-05),
('neighbourhood_Shore Acres', 2.5914205944631435e-05),
('neighbourhood_South Ozone Park', 2.5528880351360448e-05),
('neighbourhood_Ridgewood', 2.4842920434006797e-05),
('neighbourhood_Whitestone', 2.226274239934474e-05),
('neighbourhood_East New York', 2.0989208327230004e-05),
('neighbourhood_Castleton Corners', 2.065408969590442e-05),
('neighbourhood_Red Hook', 1.9939099689533358e-05),
('neighbourhood_Jamaica Hills', 1.957676571000344e-05),
('neighbourhood_Rockaway Beach', 1.8820410218523185e-05),
('neighbourhood_Sunnyside', 1.832652558892923e-05),
('neighbourhood_Williamsbridge', 1.7736198349903716e-05),
('neighbourhood_Marble Hill', 1.7023380116985254e-05),
('neighbourhood_Throgs Neck', 1.6568134039627082e-05),
('neighbourhood_Wakefield', 1.5929801262811078e-05),
('neighbourhood_Claremont Village', 1.5118466544723678e-05),
('neighbourhood_Springfield Gardens', 1.5092787712506131e-05),
('neighbourhood_Corona', 1.5014209062450137e-05),
('neighbourhood_Inwood', 1.4482967388547027e-05),
('neighbourhood_Bayside', 1.4432358116248942e-05),
('neighbourhood_Coney Island', 1.4187984615391694e-05),
('neighbourhood_Longwood', 1.402336996951179e-05),
```

```
('neighbourhood_Grymes Hill', 1.3858448143069973e-05),
('neighbourhood_Glendale', 1.3788381339454037e-05),
('neighbourhood_Kingsbridge', 1.349651667273332e-05),
('neighbourhood_Holliswood', 1.3367383125457559e-05),
('neighbourhood_Kew Gardens', 1.3283382003391169e-05),
('neighbourhood_Parkchester', 1.3056183862404494e-05),
('neighbourhood_Maspeth', 1.257230006581377e-05),
('neighbourhood_Midwood', 1.2515825350487991e-05),
('neighbourhood_Fresh Meadows', 1.2477958305459024e-05),
('neighbourhood_East Elmhurst', 1.1970542604975076e-05),
('neighbourhood_South Beach', 1.1831092799965131e-05),
('neighbourhood_Cambria Heights', 1.1240843040796888e-05),
('neighbourhood_Pelham Bay', 1.1073449913180971e-05),
('neighbourhood_Eastchester', 1.0663379725185858e-05),
('neighbourhood_Gravesend', 1.0475375831607843e-05),
('neighbourhood_Port Richmond', 9.676370726938333e-06),
('neighbourhood_Two Bridges', 9.530569229336525e-06),
('neighbourhood_Morrisania', 8.944759388622825e-06),
('neighbourhood_Borough Park', 8.688674098321856e-06),
('neighbourhood_Queens Village', 7.92330491469661e-06),
('neighbourhood_Concourse', 7.833484652662804e-06),
('neighbourhood_West Brighton', 7.752012185609397e-06),
('neighbourhood_Belle Harbor', 7.720096341327743e-06),
('neighbourhood_Willowbrook', 7.340161324755714e-06),
('neighbourhood_Port Morris', 6.912398107428829e-06),
('neighbourhood_Laurelton', 6.7346418313722656e-06),
('neighbourhood_group_Staten Island', 6.693171161086646e-06),
('neighbourhood_Oakwood', 6.432383911226699e-06),
('neighbourhood_Kew Gardens Hills', 6.412999404508163e-06),
('neighbourhood_Mill Basin', 6.370197197169665e-06),
('neighbourhood_Brownsville', 6.23021909834312e-06),
('neighbourhood_Howard Beach', 5.98935258958274e-06),
('neighbourhood_Mount Hope', 5.922324133635531e-06),
('neighbourhood_St. George', 5.872424304224148e-06),
('neighbourhood_Eltingville', 5.716875279554171e-06),
('neighbourhood_Stapleton', 5.64470731343662e-06),
('neighbourhood_Middle Village', 5.620516716944505e-06),
('neighbourhood_Fordham', 5.4632839576182265e-06),
('neighbourhood_Navy Yard', 5.272706779285579e-06),
('neighbourhood_Morris Heights', 4.980391921631419e-06),
('neighbourhood_Edgemere', 4.924540824613926e-06),
('neighbourhood_Belmont', 4.675668579752081e-06),
('neighbourhood_Norwood', 4.475129443563713e-06),
('neighbourhood_Dyker Heights', 4.329586560897639e-06),
('neighbourhood_East Morrisania', 4.1202209386279075e-06),
('neighbourhood_Hollis', 3.6079177172460504e-06),
('neighbourhood_Woodhaven', 3.5550524783476785e-06),
```

```
('neighbourhood_Ozone Park', 3.552020605694573e-06),
('neighbourhood_Bay Terrace', 3.0309969493875783e-06),
('neighbourhood_Bensonhurst', 2.813776093292632e-06),
('neighbourhood_Schuylerville', 2.8124605195782306e-06),
('neighbourhood_Columbia St', 2.7306156115518125e-06),
('neighbourhood_Highbridge', 2.583575082099338e-06),
('neighbourhood_Edenwald', 2.5388610313025785e-06),
('neighbourhood_Rosedale', 2.427360839026251e-06),
('neighbourhood_Tompkinsville', 2.388430560114815e-06),
('neighbourhood_Bellerose', 2.3761840078294175e-06),
('neighbourhood_University Heights', 2.3280585065194153e-06),
('neighbourhood_Fort Hamilton', 2.273870449466986e-06),
('neighbourhood_Breezy Point', 2.2231249249708985e-06),
('neighbourhood_Rosebank', 2.0862896800909032e-06),
('neighbourhood_North Riverdale', 1.8384413429421432e-06),
('neighbourhood_Olinville', 1.5458910790497202e-06),
('neighbourhood_Huguenot', 1.5454531918341184e-06),
('neighbourhood_Manhattan Beach', 1.4617624219518597e-06),
('neighbourhood_Bergen Beach', 1.4580807306372344e-06),
('neighbourhood_Concourse Village', 1.3070364296253393e-06),
('neighbourhood_Concord', 1.2305721158510687e-06),
('neighbourhood_Bayswater', 1.1041733498573694e-06),
('neighbourhood_Great Kills', 1.0474581698948864e-06),
('neighbourhood_Bath Beach', 1.0067005825016811e-06),
('neighbourhood_Clifton', 9.668593222864774e-07),
('neighbourhood_Morris Park', 9.645367956719146e-07),
('neighbourhood_Castle Hill', 9.08644170584847e-07),
('neighbourhood_Hunts Point', 8.5794711922469e-07),
('neighbourhood_City Island', 8.118084851997995e-07),
('neighbourhood_West Farms', 7.716613667119423e-07),
('neighbourhood_New Springville', 7.388449955270071e-07),
('neighbourhood_Fieldston', 6.778250291741982e-07),
('neighbourhood_College Point', 6.593979634368042e-07),
('neighbourhood_Emerson Hill', 6.528309291573582e-07),
('neighbourhood_Van Nest', 6.50889463555702e-07),
('neighbourhood_Soundview', 6.251055449564826e-07),
('neighbourhood_Mount Eden', 6.215180777598947e-07),
('neighbourhood_Mariners Harbor', 5.783930283592424e-07),
('neighbourhood_Dongan Hills', 5.61085359197633e-07),
('neighbourhood_Midland Beach', 4.704106645825968e-07),
('neighbourhood_Bay Terrace, Staten Island', 4.468441569687091e-07),
('neighbourhood_Melrose', 3.928063712368356e-07),
('neighbourhood_Unionport', 3.7725750442628005e-07),
('neighbourhood_Howland Hook', 3.417542515271092e-07),
('neighbourhood_Bronxdale', 3.051525048544916e-07),
('neighbourhood_Douglaston', 2.6568813870861465e-07),
('neighbourhood_Graniteville', 2.49060283836387e-07),
```

```
 ('neighbourhood_Woodlawn', 2.2967453210606673e-07),
 ('neighbourhood_Tremont', 2.2838739956785553e-07),
 ('neighbourhood_New Dorp Beach', 1.9506202300466033e-07),
 ('neighbourhood_Lighthouse Hill', 1.5838324888505038e-07),
 ('neighbourhood_Westerleigh', 1.5145682490114978e-07),
 ('neighbourhood_Grant City', 1.269049374694919e-07),
 ('neighbourhood_Baychester', 1.142089060968182e-07),
 ('neighbourhood_Tottenville', 3.484476856501777e-08),
 ('neighbourhood_New Dorp', 3.285435576179533e-08),
 ("neighbourhood_Bull's Head", 2.9618373334762254e-08),
 ('neighbourhood_Arden Heights', 2.7396015542050758e-08),
 ('neighbourhood_Co-op City', 2.2961391177121685e-08),
 ('neighbourhood_Silver Lake', 1.271651276213628e-08),
 ('neighbourhood_Little Neck', 9.605706010225654e-09),
 ('neighbourhood_Rossville', 8.72463848616149e-09),
 ('neighbourhood_New Brighton', 1.6983119406941504e-09),
 ('neighbourhood_Fort Wadsworth', 0.0),
 ('neighbourhood_Neponsit', 0.0),
 ('neighbourhood_Richmondtown', 0.0),
 ('neighbourhood_Woodrow', 0.0)]
```

Interestingly, the model assigns a high feature importance to the latitudes and longitudes, and a much lower importance to the neighbourhood information. This means that the model extracts more meaningful information from the "raw" data (as previously discussed), and makes little to no use of the neighbourhood information.

Considering that the majority of the columns in our dataset has been generated from neighbourhood information (by 1-hot encoding the original columns), it is reasonable to discard those columns altogether. By doing that, the random forest will not have as many "noisy" features to select from at each split. As a consequence it will be more likely that, at each split, more useful features will be available, thus building better trees.

We can redo the entire preprocessing step as follows:

[15]:
```
# Only encode "room_type"
df_1h = pd.get_dummies(df, columns=['room_type'])

# discard "neighbourhood" and "neighbourhood_group"
df_dropped = df_1h.
 ↪drop(columns=["neighbourhood_group","neighbourhood","host_id", "name",␣
 ↪"host_name", "last_review"])
train_valid_mask = ~df_dropped["price"].isna()
feature_names = df_dropped[train_valid_mask].drop(columns=["price"]).columns

X = df_dropped.drop(columns=["price"]).values
y = df_dropped["price"].values

X_train_valid = X[train_valid_mask]
```

```
y_train_valid = y[train_valid_mask]
X_test = X[~train_valid_mask]
y_test = y[~train_valid_mask]

X_train, X_valid, y_train, y_valid = train_test_split(X_train_valid,␣
 ↪y_train_valid, shuffle=True, random_state=42)

reg = RandomForestRegressor(100, random_state=42)
reg.fit(X_train, y_train)
r2_score(y_valid, reg.predict(X_valid))
```

[15]: 0.10485905409912633

We can immediately see that there is an improvement in R^2 score. We are also significantly reducing the number of features (from 236 down to 10), so the decision of discarding the neighbourhood information is particularly useful.

[16]:
```
sorted(zip(feature_names, reg.feature_importances_), key=lambda x: x[1],␣
 ↪reverse=True)
```

[16]:
```
[('longitude', 0.3106356538341069),
 ('latitude', 0.2432212751382898),
 ('availability_365', 0.10206173081645416),
 ('minimum_nights', 0.09453420146707416),
 ('reviews_per_month', 0.07067698724606092),
 ('room_type_Entire home/apt', 0.06307473812272034),
 ('calculated_host_listings_count', 0.06097854808980696),
 ('number_of_reviews', 0.053690275059188476),
 ('room_type_Shared room', 0.0007054953430413454),
 ('room_type_Private room', 0.0004210948832570403)]
```

We can now see that the model is giving even more importance to the latitude and longitude. This is because we have removed any other source of location information.

Now, let us try to re-introduce two of the features we previously discarded: `id` and `host_id`. We will first introduce the first one, then the second one, then both together. For each configuration of features we will train a separate model and assess how it behaves. If our initial hypotheses are correct, these two features should not have a particular impact on our model's performance.

[17]:
```
for include_features in [["id"], ["host_id"], ["id", "host_id"]]:
    df_1h = pd.get_dummies(df, columns=['room_type'])

    # Extract the "id" information
    if "id" in include_features:
        df_1h["id"] = df_1h.index

    df_dropped = df_1h.drop(columns=["neighbourhood_group","neighbourhood",␣
 ↪"name", "host_name", "last_review"])
```

```
    # if "host_id" should not be kept, it is discarded
    if "host_id" not in include_features:
        df_dropped = df_dropped.drop(columns=["host_id"])
    train_valid_mask = ~df_dropped["price"].isna()
    feature_names = df_dropped[train_valid_mask].drop(columns=["price"]).columns

    X = df_dropped.drop(columns=["price"]).values
    y = df_dropped["price"].values

    X_train_valid = X[train_valid_mask]
    y_train_valid = y[train_valid_mask]
    X_test = X[~train_valid_mask]
    y_test = y[~train_valid_mask]

    X_train, X_valid, y_train, y_valid = train_test_split(X_train_valid,␣
 ↪y_train_valid, shuffle=True, random_state=42)

    reg = RandomForestRegressor(100, random_state=42)
    reg.fit(X_train, y_train)
    print(include_features, r2_score(y_valid, reg.predict(X_valid)))
```

```
['id'] 0.11644446892306404
['host_id'] 0.11778902062981622
['id', 'host_id'] 0.13278619065870456
```

[18]:
```
sorted(zip(feature_names, reg.feature_importances_), key=lambda x: x[1],␣
 ↪reverse=True)
```

[18]:
```
[('longitude', 0.2031465357932134),
 ('host_id', 0.17614377946958268),
 ('id', 0.14673197942285066),
 ('latitude', 0.13067312129464947),
 ('minimum_nights', 0.07846764247070498),
 ('availability_365', 0.07742011112858815),
 ('room_type_Entire home/apt', 0.06307473812272034),
 ('reviews_per_month', 0.04334148791331092),
 ('calculated_host_listings_count', 0.04328071480252347),
 ('number_of_reviews', 0.03700440526719194),
 ('room_type_Private room', 0.0004350265333742561),
 ('room_type_Shared room', 0.0002804577812896995)]
```

Surprisingly, both features result in a performance improvement. This makes sense when we consider the fact that both id and host_id are sequential in nature. In a way, both variables are a proxy for the moment in time when the post was created. The "creation time" of the post can contain significant information (e.g. there may be periods of the year with increased tourism, or gentrification-related neighbourhood improvements).

Based on the available dataset, there is a quick way for us to validate the possibility that the id's are actually related to the time at which the posts were created. We know the average number of ratings received by each post per month, as well as the total number of ratings received for each apartment. We can work out the number of months a post has been published as `number_of_reviews/reviews_per_month`.

[19]:
```
months = (df_1h["number_of_reviews"] / df_1h["reviews_per_month"])
months = months.fillna(months.mean()) # fill NA values with the mean value
```

We can then assess the correlation (in terms of Pearson's correlation coefficient) between this new feature and `id`, `host_id`

[20]:
```
import numpy as np

np.corrcoef([months, df.index])[0][1], np.corrcoef([months, df["host_id"]])[0][1]
```

[20]:
```
(-0.7861196186270615, -0.4432558467997288)
```

Both values are negatively correlated: indeed, a lower `id/host_id` corresponds to an older post (i.e. higher number of months). The correlation in existance is higher with the actual `id` of the post. The `host_id`'s relationship is weaker: clearly, older posts can only have been created by "older" users (an "older" user is a user who has been registered to the website for longer), but that is as far as this relationship can go.

We also know that many of the entries in our dataset have received no reviews. For all these entries, it will be impossible for us to work out the post's age. More specifically, we can compute the fraction of posts with no reviews as follows:

[21]:
```
len( df_1h[df_1h["number_of_reviews"] == 0])/len(df_1h)
```

[21]:
```
0.20558339298496778
```

Approximately 20% of the entries in our dataset cannot be assigned a "timestamp", if not through their id. On the other hand, we do have the timestamp information, in terms of id, for all entries. For this reason, we will be keeping both `id` and `host_id` in our dataset. Do keep in mind, though, that in many (most) cases, ids will not be informative in the least (especially when you also have reliable information on creation dates, or when ids are generated randomly).

Finally, we can consider exploring the title of posts (`name` attribute). This is a natural language string which may contain useful information we have been sitting on this whole time.

We can process the textual data using sklearn's `TfidfVectorizer`, which will split each title into tokens and remove stopwords. We can consider using a binary feature for each of the most popular words, since we will only consider the presence/absence of terms as relevant (binary term frequency, with no inverse document frequency).

[22]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(stop_words="english", binary=True, use_idf=False,␣
 ↪norm=False)
```

```
# word presence matrix (i-th row, j-th col => 1 if j-th word is contained in␣
↪i-th title)
wpm = vectorizer.fit_transform(df_1h["name"].fillna(""))
```

We can now take the N most popular words and add a boolean feature for each of them. For example, with N = 150, we get the following list of words.

[23]:
```
N = 150
freq = sorted(zip(vectorizer.get_feature_names(), wpm.sum(axis=0).tolist()[0]),␣
↪key=lambda x: x[1], reverse=True)[:N]
freq
```

[23]:
```
[('room', 10213.0),
 ('bedroom', 8145.0),
 ('private', 7275.0),
 ('apartment', 6758.0),
 ('cozy', 5093.0),
 ('apt', 4727.0),
 ('brooklyn', 4174.0),
 ('studio', 4105.0),
 ('spacious', 3796.0),
 ('manhattan', 3585.0),
 ('park', 3136.0),
 ('east', 3086.0),
 ('sunny', 2945.0),
 ('williamsburg', 2742.0),
 ('beautiful', 2510.0),
 ('near', 2373.0),
 ('village', 2343.0),
 ('nyc', 2283.0),
 ('loft', 2092.0),
 ('large', 2080.0),
 ('heart', 2070.0),
 ('bed', 2044.0),
 ('modern', 1821.0),
 ('central', 1815.0),
 ('bright', 1724.0),
 ('luxury', 1710.0),
 ('home', 1607.0),
 ('west', 1594.0),
 ('1br', 1576.0),
 ('new', 1568.0),
 ('location', 1561.0),
 ('bushwick', 1438.0),
 ('charming', 1386.0),
 ('upper', 1354.0),
 ('midtown', 1283.0),
```

```
('br', 1271.0),
('quiet', 1249.0),
('brownstone', 1194.0),
('clean', 1163.0),
('great', 1154.0),
('harlem', 1144.0),
('square', 1109.0),
('close', 1062.0),
('bath', 1035.0),
('subway', 1021.0),
('garden', 998.0),
('huge', 979.0),
('heights', 945.0),
('times', 880.0),
('prime', 856.0),
('duplex', 852.0),
('min', 833.0),
('city', 817.0),
('amazing', 799.0),
('house', 779.0),
('2br', 776.0),
('train', 752.0),
('view', 748.0),
('chelsea', 728.0),
('suite', 715.0),
('lovely', 710.0),
('renovated', 700.0),
('space', 671.0),
('bathroom', 665.0),
('big', 665.0),
('soho', 624.0),
('york', 622.0),
('best', 621.0),
('astoria', 619.0),
('comfortable', 613.0),
('comfy', 611.0),
('floor', 607.0),
('hill', 606.0),
('slope', 605.0),
('gorgeous', 601.0),
('entire', 597.0),
('prospect', 593.0),
('jfk', 583.0),
('greenpoint', 559.0),
('kitchen', 538.0),
('place', 526.0),
('views', 520.0),
```

```
('perfect', 513.0),
('mins', 507.0),
('townhouse', 497.0),
('balcony', 489.0),
('artist', 479.0),
('away', 470.0),
('minutes', 469.0),
('backyard', 468.0),
('building', 464.0),
('gym', 462.0),
('doorman', 451.0),
('uws', 451.0),
('cute', 448.0),
('ny', 445.0),
('oasis', 440.0),
('shared', 439.0),
('queens', 438.0),
('queen', 435.0),
('rooftop', 433.0),
('bk', 427.0),
('lower', 425.0),
('sonder', 423.0),
('nice', 422.0),
('stuy', 417.0),
('terrace', 414.0),
('furnished', 413.0),
('historic', 413.0),
('bdrm', 408.0),
('light', 403.0),
('penthouse', 403.0),
('15', 400.0),
('downtown', 394.0),
('chic', 393.0),
('stylish', 387.0),
('les', 383.0),
('filled', 379.0),
('sq', 376.0),
('steps', 373.0),
('street', 368.0),
('convenient', 363.0),
('living', 359.0),
('newly', 357.0),
('ues', 357.0),
('lga', 348.0),
('gem', 347.0),
('stay', 344.0),
('crown', 342.0),
```

```
('master', 331.0),
('columbia', 323.0),
('family', 322.0),
('area', 319.0),
('bedstuy', 314.0),
('block', 309.0),
('sun', 303.0),
('clinton', 302.0),
('friendly', 301.0),
('walk', 301.0),
('condo', 297.0),
('20', 295.0),
('center', 293.0),
('st', 293.0),
('brand', 290.0),
('stunning', 285.0),
('3br', 284.0),
('greene', 280.0),
('bedrooms', 279.0),
('patio', 277.0),
('time', 276.0)]
```

As you can see, the most frequent words can also be quite useful for extracting additional information such as spaces available (e.g. balcony, backyard), place conditions (e.g. renovated, modern, sunny) and even information on the possible price ranges (e.g. luxury).

Our `wpm` matrix already contains binary values based on whether words are present or not. We can use this matrix, with only the N most frequent columns, to build an additional DataFrame to be attached to the original one.

[24]:
```
import numpy as np
# mask to be used to filter columns in wpm (only keeps the ones for the 100 most␣
↪frequent words)
words = [ word for word, _ in freq ]
mask = [ w in words for w in vectorizer.get_feature_names() ]
words_ = [ w for w in vectorizer.get_feature_names() if w in words ]
words_df = pd.DataFrame(data=wpm[:, np.array(mask)].toarray(),␣
↪columns=[f"word_{word}" for word in words_], index=df_1h.index)
```

[25]:
```
# Only encode "room_type"
df_1h = pd.get_dummies(df, columns=['room_type'])

df_1h = df_1h.join(pd.DataFrame(data=wpm[:, np.array(mask)].toarray(),␣
↪columns=[f"word_{word}" for word in words_], index=df_1h.index))

# discard "neighbourhood" and "neighbourhood_group"
df_dropped = df_1h.drop(columns=["neighbourhood_group","neighbourhood", "name",␣
↪"host_name", "last_review"])
```

```
df_dropped["id"] = df_dropped.index
train_valid_mask = ~df_dropped["price"].isna()
feature_names = df_dropped[train_valid_mask].drop(columns=["price"]).columns

X = df_dropped.drop(columns=["price"]).values
y = df_dropped["price"].values

X_train_valid = X[train_valid_mask]
y_train_valid = y[train_valid_mask]
X_test = X[~train_valid_mask]
y_test = y[~train_valid_mask]

X_train, X_valid, y_train, y_valid = train_test_split(X_train_valid,␣
↪y_train_valid, shuffle=True, random_state=42)

reg = RandomForestRegressor(100, random_state=42)
reg.fit(X_train, y_train)
r2_score(y_valid, reg.predict(X_valid))
```

[25]: 0.16344001987598977

We could continue with the preprocessing step by (1) studying how having larger or smaller N affects the performance, or (2) introducing more meaningful weights for each word (e.g. actual tf-idf), (3) trying new approaches on other features (e.g. the introduction of polynomial features), (4) introducing new datasets (e.g. with points of interest close to each place) and so on.

For the sake of brevity, we will be stopping the preprocessing here, and proceed with a hyperparamter tuning step. We will be using a random forest regressor, but you may consider trying other regressors and assess their performance.

We will be defining a small subset of possible hyperparameters for our grid search, once again you may find better configurations that has not been explored here.

For the grid search, we will be using 5-fold cross validation. We have already defined a validation set for previous purposes, but we have already been using it to assess the model performance. With cross-validation, we can make sure that we do not overfit a single subset of data.

[26]:
```
from sklearn.model_selection import GridSearchCV

param_grid = {
    "n_estimators": [100, 250, 500],
    "criterion": ["mse", "mae"],
    "max_features": ["auto", "sqrt", "log2"],
    "random_state": [42], # always use the samet random seed
    "n_jobs": [-1], # for parallelization
}

gs = GridSearchCV(RandomForestRegressor(), param_grid, scoring="r2", n_jobs=-1,␣
↪cv=5)
```

```
gs.fit(X_train_valid, y_train_valid)
gs.best_score_
```

[26]: 0.22750076738532582

Since GridSearchCV already refits the best model with the entire dataset (if refit=True, which is the default value), we can use gs to predict the prices for X_test. Then, we can generate a csv file with the predicted values.

[27]:
```
y_pred = gs.predict(X_test)
pd.DataFrame(y_pred, index=df[~train_valid_mask].index).to_csv("output.csv",␣
↪index_label="Id", header=["Predicted"])
```

By submitting output.csv to the submission platform, we get the following results:

- Public: 0.3083
- Private: 0.2049

As you can see, there is a large difference in R2 score obtained for the two sets. This could be an indication of overfitting, since we are performing much better on the public set than we are on the private set. However, we have not used the public set for any of the decisions made so far (we have only used a validation set or cross-validation, both of which are parts of the development set). We can see how the private score is closer to the best score we obtained with the grid search. This is likely an indicator that the data in the public and the private sets are not sampled from the same distribution, and that the private set is more similar to the data we used for the training.

For future competitions, we will make sure that this kind of divergence in the sampling of the two sets will no longer occur.