Table of Contents

Introduction:

Since 2008, guests and hosts have used Airbnb to expand on traveling possibilities and present a more unique, personalized way of experiencing the world. Today, Airbnb became one of a kind service that is used and recognized by the whole world. Data analysis on millions of listings provided through Airbnb is a crucial factor for the company. These millions of listings generate a lot of data - data that can be analyzed and used for security, business decisions, understanding of customers' and providers' (hosts) behavior and performance on the platform, guiding marketing initiatives, implementation of innovative additional services and much more.

The dataset is sourced from a Kaggle competition "New York City Airbnb Open Data"(https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data).

Objectives:

The main objective is to perform descriptive and exploratory analysis of the data, in order to understand how each variable behave individually.

Compare different linear regression models to predict the price of a listing hosted.

Data Dictionary:

```
print(data.columns.tolist())
```

```
['id', '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_36
5']
```

Variable	Definition
id	Listing ID
name	name of listing
host_id	Host ID
host_name	Name of the host
neighbourhood_group	Location
neighbourhood	Area
latitude	Latitude Coordinates
longitude	Longitude Coordinates
room_type	Listing Space Type
price	Price in Dollars
minimum_nights	Minimum number of nights booked
number_of_reviews	Number of reviews received
last_review	Date of Last review received
reviews_per_month	Reviews per month received
calculated_host_listings_count	Count of host listings
availability_365	Days of Availability in a year

Data information:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48895 entries, 0 to 48894
Data columns (total 16 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   id                              48895 non-null  int64  
 1   name                            48879 non-null  object 
 2   host_id                         48895 non-null  int64  
 3   host_name                       48874 non-null  object 
 4   neighbourhood_group             48895 non-null  object 
 5   neighbourhood                   48895 non-null  object 
 6   latitude                        48895 non-null  float64
 7   longitude                       48895 non-null  float64
 8   room_type                       48895 non-null  object 
 9   price                           48895 non-null  int64  
 10  minimum_nights                  48895 non-null  int64  
 11  number_of_reviews               48895 non-null  int64  
 12  last_review                     38843 non-null  object 
 13  reviews_per_month               38843 non-null  float64
 14  calculated_host_listings_count  48895 non-null  int64  
 15  availability_365                48895 non-null  int64  
dtypes: float64(3), int64(7), object(6)
memory usage: 6.0+ MB
```

We have 6 categorical variables and 10 numerical variables.

```
data.dtypes.value_counts()
```

```
int64      7
object     6
float64    3
dtype: int64
```

Summary Statistics:

```
data.describe().T
```

	count	mean	std	min	25%	50%	75%	max
id	48895.0	1.901714e+07	1.098311e+07	2539.00000	9.471945e+06	1.967728e+07	2.915218e+07	3.648724e+07
host_id	48895.0	6.762001e+07	7.861097e+07	2438.00000	7.822033e+06	3.079382e+07	1.074344e+08	2.743213e+08
latitude	48895.0	4.072895e+01	5.453008e-02	40.49979	4.069010e+01	4.072307e+01	4.076311e+01	4.091306e+01
longitude	48895.0	-7.395217e+01	4.615674e-02	-74.24442	-7.398307e+01	-7.395568e+01	-7.393627e+01	-7.371299e+01
price	48895.0	1.527207e+02	2.401542e+02	0.00000	6.900000e+01	1.060000e+02	1.750000e+02	1.000000e+04
minimum_nights	48895.0	7.029962e+00	2.051055e+01	1.00000	1.000000e+00	3.000000e+00	5.000000e+00	1.250000e+03
number_of_reviews	48895.0	2.327447e+01	4.455058e+01	0.00000	1.000000e+00	5.000000e+00	2.400000e+01	6.290000e+02
reviews_per_month	38843.0	1.373221e+00	1.680442e+00	0.01000	1.900000e-01	7.200000e-01	2.020000e+00	5.850000e+01
calculated_host_listings_count	48895.0	7.143982e+00	3.295252e+01	1.00000	1.000000e+00	1.000000e+00	2.000000e+00	3.270000e+02
availability_365	48895.0	1.127813e+02	1.316223e+02	0.00000	0.000000e+00	4.500000e+01	2.270000e+02	3.650000e+02

```
data.describe(include='object')
```

	name	host_name	neighbourhood_group	neighbourhood	room_type	last_review
count	48879	48874	48895	48895	48895	38843
unique	47905	11452	5	221	3	1764
top	Hillside Hotel	Michael	Manhattan	Williamsburg	Entire home/apt	2019-06-23
freq	18	417	21661	3920	25409	1413

Percentage of missing values in the columns:

```
(data.isna().sum() *100) / data.shape[0]
```

```
id                                0.000000
name                              0.032723
host_id                           0.000000
host_name                         0.042949
neighbourhood_group               0.000000
neighbourhood                     0.000000
latitude                          0.000000
longitude                         0.000000
room_type                         0.000000
price                             0.000000
minimum_nights                    0.000000
number_of_reviews                 0.000000
last_review                      20.558339
reviews_per_month                20.558339
calculated_host_listings_count    0.000000
availability_365                  0.000000
dtype: float64
```

Exploratory Data Analysis:

- Observations from the data description :
  - “id”, "name" and "host_name" are irrelevant and insignificant to our data analysis as they will not add any predictive value to the model.
  - Columns "last_review" and "review_per_month" have a few null values and will need simple handling.
  - "last_review" is date; if there were no reviews for the listing - date will not exist.
  - For "review_per_month" column we can simply append it with 0.0 for missing values; we can see that in "number_of_review" that column will have a 0, therefore following this logic with 0 total reviews there will be 0.0 rate of reviews per month.

```
data.drop(['id','name','host_name','last_review'], axis=1, inplace=True)
data.fillna({'reviews_per_month':0}, inplace=True)
```

- We have 5 neighbourhood groups, 3 room types and 221 neighbourhoods in total.

```
data.neighbourhood_group.unique()
```

```
array(['Brooklyn', 'Manhattan', 'Queens', 'Staten Island', 'Bronx'],
      dtype=object)
```

```
data.room_type.unique()
```

```
array(['Private room', 'Entire home/apt', 'Shared room'], dtype=object)
```

```
len(data.neighbourhood.unique())
```

```
221
```

- Since we have a big list of Host ID, we will have a look at how the top 15 hosts with the most listings on Air Bnb.

```
len(data.host_id.unique())
```

```
37457
```

```
data.host_id.value_counts().head(15)
```

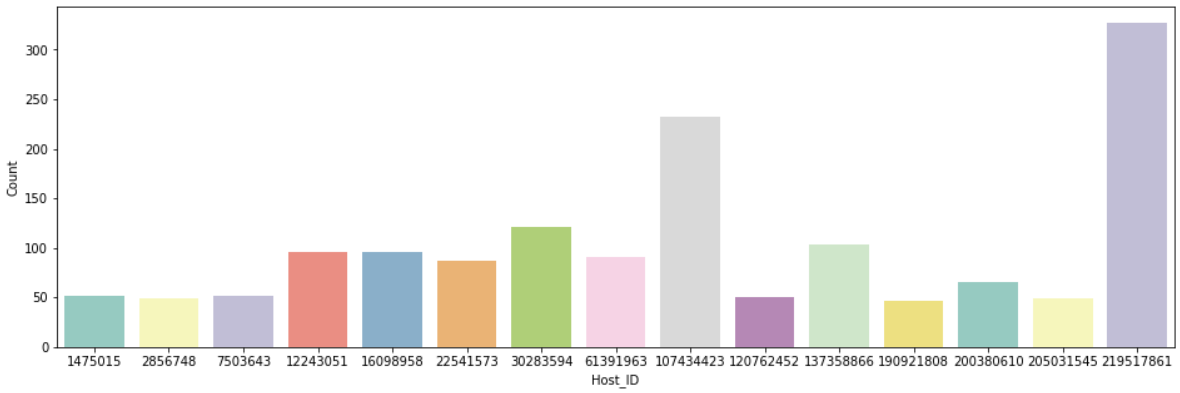
```
219517861    327
107434423    232
30283594     121
137358866    103
12243051      96
16098958      96
61391963      91
22541573      87
200380610     65
7503643       52
1475015       52
120762452     50
2856748       49
205031545     49
190921808     47
Name: host_id, dtype: int64
```

We can verify if the host listing count from the already available field 'calculated_host_listings_count' as:

```
data.calculated_host_listings_count.max()
```

```
327
```

Upon Visualizing, we see that the hosts have a good distribution in their listings:



- Looking at the listing in the neighbourhood by their 'Price', we observed that Manhattan has the highest range of prices for the listings with $150 price on an average, with Brooklyn at $90 per night. Queens and Staten Island appeared to have very similar distributions, while Bronx is the cheapest of them all.

Stats	Brooklyn	Manhattan	Queens	Staten Island	Bronx
min	0.0	0.0	10.0	13.0	0.0
25%	60.0	95.0	50.0	50.0	45.0
50%	90.0	150.0	75.0	75.0	65.0
75%	150.0	220.0	110.0	110.0	99.0
max	10000.0	10000.0	10000.0	5000.0	2500.0

- Just like Host IDs, due to the number of neighbourhoods being large, we will look at the top 15 neighbourhoods with the largest listings.

```
data.neighbourhood.value_counts().head(15)
```

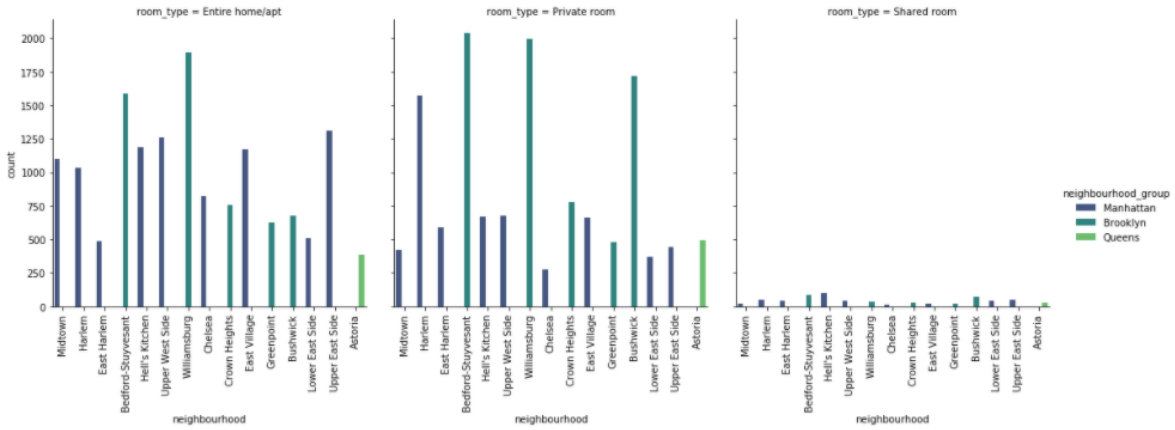
```
Williamsburg          3920
Bedford-Stuyvesant    3714
Harlem                2658
Bushwick              2465
Upper West Side       1971
Hell's Kitchen        1958
East Village          1853
Upper East Side       1798
Crown Heights         1564
Midtown               1545
East Harlem           1117
Greenpoint            1115
Chelsea               1113
Lower East Side        911
Astoria                900
Name: neighbourhood, dtype: int64
```

  - Looking at the listings in these neighbourhoods by the different ‘room_type’ and also looking the groups they belong to, we can say that 'Shared room' type Airbnb listing is barely available among the top neighbourhoods. Also that ‘Queens’ doesn’t show up a lot in them as well, this may be due to how famous Manhattan and Brooklyn are as tourist spots.

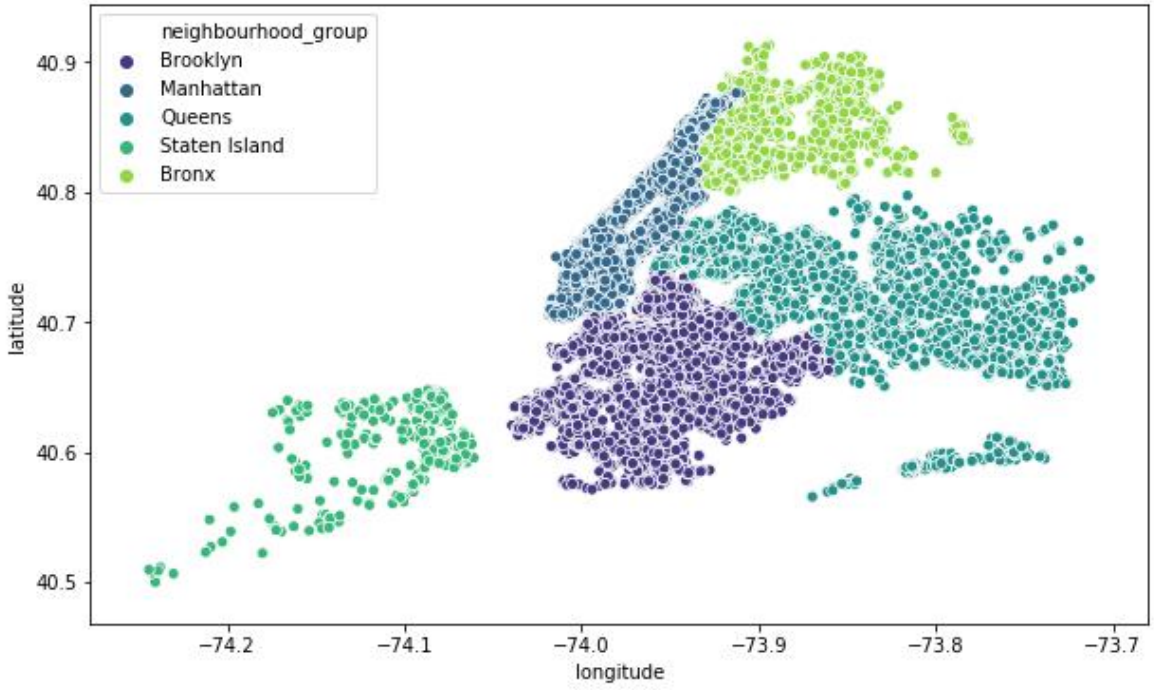
```
top_nb = data.loc[data['neighbourhood'].isin(top_nb_df['neighbourhoods'])]
```

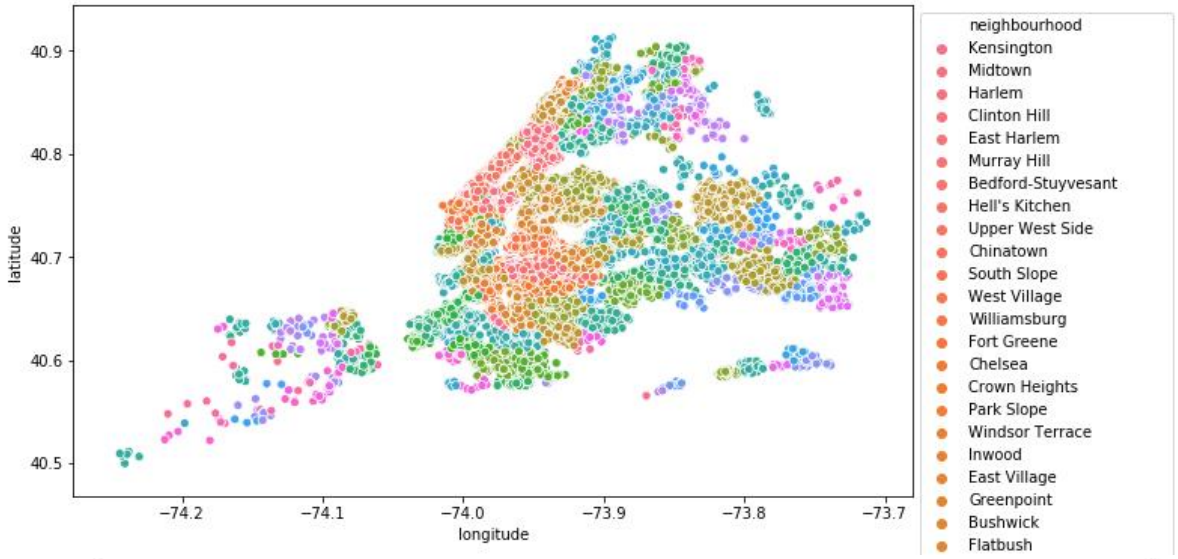
```
fig=sns.catplot(x='neighbourhood', hue='neighbourhood_group', col='room_type', data=top_nb, kind='count',palette="viridis")
fig.set_xticklabels(rotation=90)
```

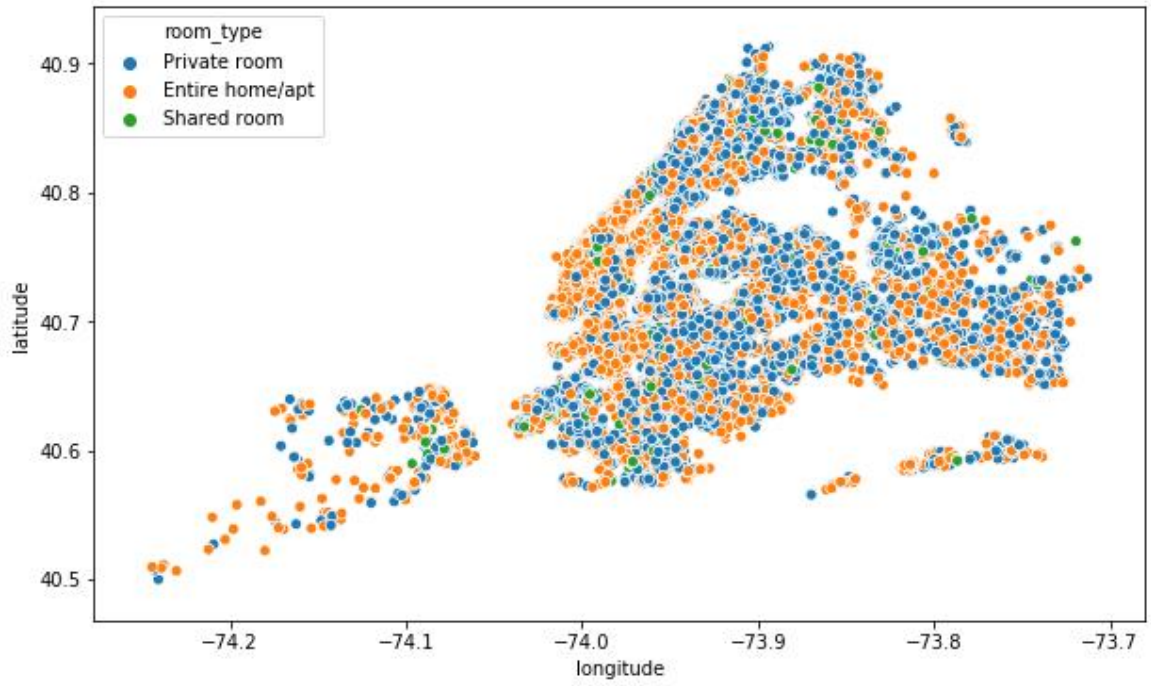
```
<seaborn.axisgrid.FacetGrid at 0x224bedea908>
```

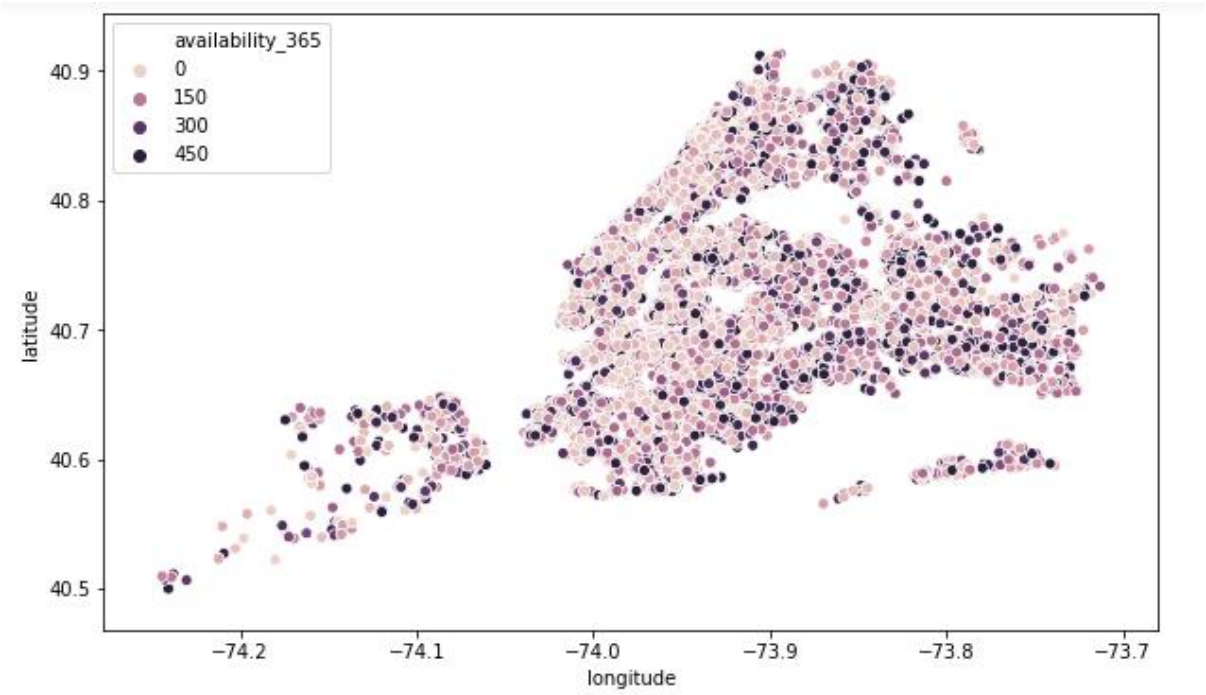


- 'Longitude' and 'Latitude' allows us to visualize NYC by neighbourhood, neighbourhood_group, room_type and their availability.




neighbourhood
Kensington
Midtown
Harlem
Clinton Hill
East Harlem
Murray Hill
Bedford-Stuyvesant
Hell's Kitchen
Upper West Side
Chinatown
South Slope
West Village
Williamsburg
Fort Greene
Chelsea
Crown Heights
Park Slope
Windsor Terrace
Inwood
East Village
Greenpoint
Bushwick
Flatbush
latitude
longitude
40.9
40.8
40.7
40.6
40.5
−74.2
−74.1
−74.0
−73.9
−73.8
−73.7


room_type
Private room
Entire home/apt
Shared room
latitude
longitude
40.9
40.8
40.7
40.6
40.5
−74.2
−74.1
−74.0
−73.9
−73.8
−73.7



- Looking at the listings with most number of reviews and most reviews per month, we observe that most listings are made for 'Private Room' with average price being $66 and $72 respectively.

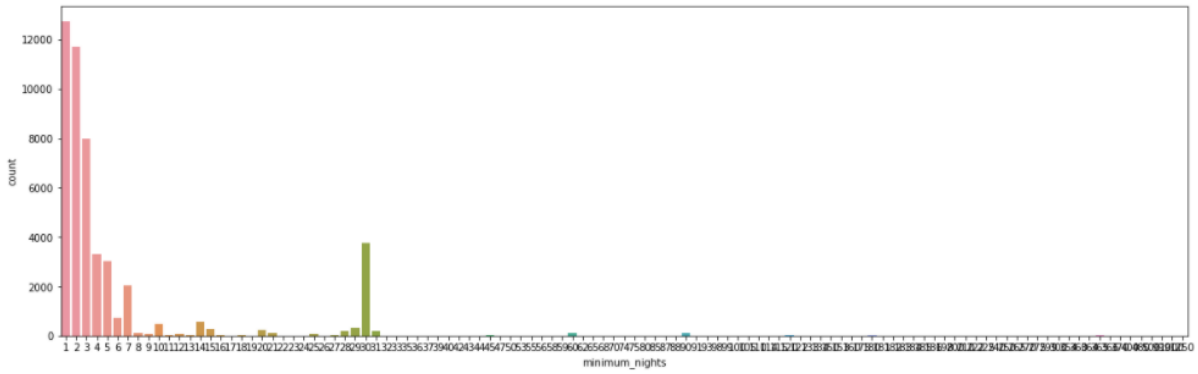
```
x= data.nlargest(15,'reviews_per_month')
print(x.price.mean())
print(x.room_type.value_counts())
```

```
72.46666666666667
Private room       14
Entire home/apt     1
Name: room_type, dtype: int64
```

```
y=data.nlargest(15,'number_of_reviews')
print(y.price.mean())
print(y.room_type.value_counts())
```

```
66.6
Private room       13
Entire home/apt     2
Name: room_type, dtype: int64
```

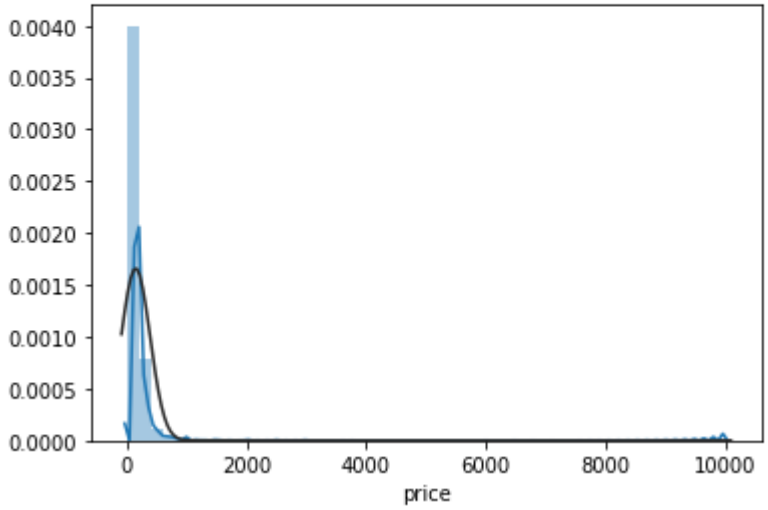
- The distribution of 'minimum_nights' indicate that most listings have a minimum booking date between 1 and 3.



- Checking normality of the 'Price'.

```
plt.figure(figsize=(6,4))
sns.distplot(data['price'], fit=norm)
```

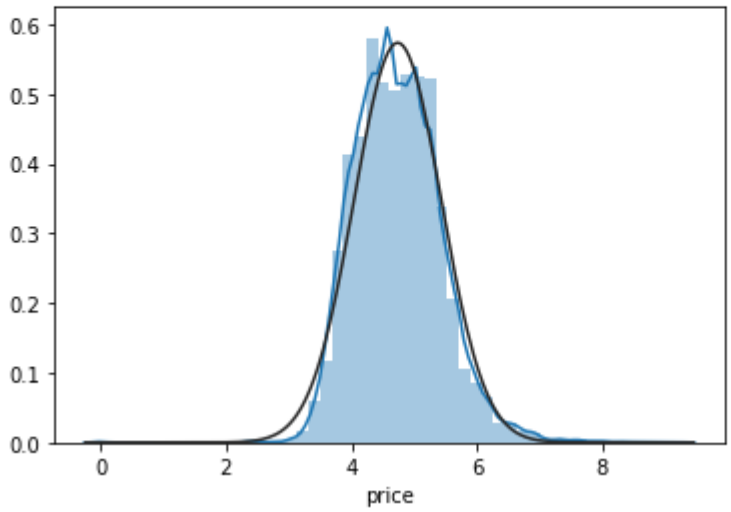
```
<matplotlib.axes._subplots.AxesSubplot at 0x224c40894c8>
```



Since 'price' is not normally distributed, it is worthwhile to apply some transformation to make it normal. Apply a log function on the price gives us a much normally distributed data.

```
log_price = np.log(data.price +1)
plt.figure(figsize=(6,4))
sns.distplot(log_price, fit=norm)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x224c55d2f88>
```



Hence we replace ‘price’ by its log value.

```
data['log_price'] = np.log(data.price +1)
data.drop(['price'], axis=1, inplace=True)
```

Model Evaluation:

For the first step we employ to One Hot Encoder and Label Encoder for the Categorical variables – room_type, neighbourhood and neighbourhood_group.

One Hot Encoding

```
mask = data.dtypes == np.object
categorical_cols = data.columns[mask]
```

```
num_ohc_cols = (data[categorical_cols]
                .apply(lambda x: x.nunique())
                .sort_values(ascending=False))
```

```
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
# Copy of the data
data_ohc = data.copy()
# The encoders
le = LabelEncoder()
ohc = OneHotEncoder()
# ohc = OneHotEncoder(drop='first')

for col in num_ohc_cols.index:

    # Integer encode the string categories
    dat = le.fit_transform(data_ohc[col]).astype(np.int)

    # Remove the original column from the dataframe
    data_ohc = data_ohc.drop(col, axis=1)

    # One hot encode the data--this returns a sparse array
    new_dat = ohc.fit_transform(dat.reshape(-1,1))

    # Create unique column names
    n_cols = new_dat.shape[1]
    col_names = ['_'.join([col, str(x)]) for x in range(n_cols)]

    # Create the new dataframe
    new_df = pd.DataFrame(new_dat.toarray(),
                          index=data_ohc.index,
                          columns=col_names)
    # Append the new data to the dataframe
    data_ohc = pd.concat([data_ohc, new_df], axis=1)
```

After this step our number of variable count increases to 266.

Our next step is to split the data in training and testing using train_test_split:

```
X = data_ohc.copy().drop('log_price', axis = 1)
y = data_ohc['log_price'].copy()
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

We define a function to calculate the root mean squared error:

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
def rmse(ytrue, ypredicted):
    return np.sqrt(mean_squared_error(ytrue, ypredicted))
```

Unfortunately, I was unable to transform the variable with Polynomial features due to system limitation, so I will proceed with only the Encoded variables.

```
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
s= StandardScaler()
X_train_s = s.fit_transform(X_train)
X_test_s = s.transform(X_test)
pf = PolynomialFeatures(degree=2, include_bias=False)
X_train_pf = pf.fit_transform(X_train_s)
X_test_pf = pf.fit_transform(X_test_s)
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-137-e2c547d76f2d> in <module>
      4 X_test_s = s.transform(X_test)
      5 pf = PolynomialFeatures(degree=2, include_bias=False)
----> 6 X_train_pf = pf.fit_transform(X_train_s)
      7 X_test_pf = pf.fit_transform(X_test_s)
```

Now we define the following regression models and fit it to our training dataset and calculate the r2 score and root mean square error with the predicted values:

a) Linear Regression
b) Ridge Regression (Cross Validation)
c) Lasso Regression (Cross Validation)
d) ElasticNet Regression (Cross Validation)

```
from sklearn.linear_model import LinearRegression

linearRegression = LinearRegression().fit(X_train, y_train)

linearRegression_rmse = rmse(y_test, linearRegression.predict(X_test))

print(linearRegression_rmse)
```

```
0.464060659716015
```

```
import warnings
warnings.filterwarnings('ignore', module='sklearn')

from sklearn.linear_model import RidgeCV

alphas = [0.005, 0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30, 80]

ridgeCV = RidgeCV(alphas=alphas,
                  cv=4).fit(X_train, y_train)

ridgeCV_rmse = rmse(y_test, ridgeCV.predict(X_test))

print(ridgeCV.alpha_, ridgeCV_rmse)
```

```
1.0 0.4639945017550402
```

```
from sklearn.linear_model import LassoCV

alphas2 = np.array([1e-5, 5e-5, 0.0001, 0.0005])

lassoCV = LassoCV(alphas=alphas2,
                  max_iter=5e4,
                  cv=3).fit(X_train, y_train)

lassoCV_rmse = rmse(y_test, lassoCV.predict(X_test))

print(lassoCV.alpha_, lassoCV_rmse)  # Lasso is slower
```

```
5e-05 0.46385457578264444
```

```
from sklearn.linear_model import ElasticNetCV

l1_ratios = np.linspace(0.1, 0.9, 9)

elasticNetCV = ElasticNetCV(alphas=alphas2,
                            l1_ratio=l1_ratios,
                            max_iter=1e4).fit(X_train, y_train)
elasticNetCV_rmse = rmse(y_test, elasticNetCV.predict(X_test))

print(elasticNetCV.alpha_, elasticNetCV.l1_ratio_, elasticNetCV_rmse)
```

```
5e-05 0.8 0.46381998412313596
```

After the training the models , we compare their Root Mean Squared Errors:

```
rmse_vals = [linearRegression_rmse, ridgeCV_rmse, lassoCV_rmse, elasticNetCV_rmse]
r2_vals = [linearRegression_r2, ridgeCV_r2, lassoCV_r2, elasticNetCV_r2]
labels = ['Linear', 'Ridge', 'Lasso', 'ElasticNet']

rmse_df = pd.Series(rmse_vals, index=labels).to_frame()
rmse_df.rename(columns={0: 'RMSE'}, inplace=1)
rmse_df['r2'] = r2_vals
rmse_df
```

	RMSE	r2
Linear	0.464061	0.544351
Ridge	0.463995	0.544480
Lasso	0.463855	0.544755
ElasticNet	0.463820	0.544823

The testing error from Ridge, Lasso and ElasticNet are similar to the baseline Linear Regression testing errors.

When we look at the regularization for their optimal values of alpha, we see that Lasso has the strongest regularization forcing many of the coefficients to be reduced to 0 while ridge almost forces none of them to 0. ElasticNet provides a well balance between the two and hence we should consider it to be the best model for this project.

```
print(np.sum(ridgeCV.coef_ != 0))
print(np.sum(lassoCV.coef_ != 0))
print(np.sum(elasticNetCV.coef_ != 0))
```

```
233
144
154
```

When we look at the main drivers of the ElasticNet model, we can see that room_type and neighbourhood had the most impact on the prices, which is expected as we observed the range of prices between neighbourhood groups during our Exploratory analysis.

```
order = np.argsort(np.abs(elasticNetCV.coef_))[::-1]
for i in order:
    coef_ = elasticNetCV.coef_[i]
    if coef_ > 0:
        print(X.columns[i] + ', ' + str(elasticNetCV.coef_[i]))
```

```
neighbourhood_167, 0.8463979059966412
room_type_0, 0.7064124341196099
neighbourhood_21, 0.5699766931599237
neighbourhood_197, 0.5093407143977304
neighbourhood_36, 0.49913787276495714
neighbourhood_82, 0.3989968920989421
neighbourhood_98, 0.3931945964159804
neighbourhood_group_2, 0.38301525992096386
neighbourhood_14, 0.3499643415396056
neighbourhood_204, 0.34668853320879445
neighbourhood_25, 0.3410618668627689
neighbourhood_53, 0.3153415015322614
neighbourhood_43, 0.3050314428099444
neighbourhood_65, 0.3011792477368236
neighbourhood_75, 0.2879824145491758
neighbourhood_214, 0.287181726100275
neighbourhood_158, 0.28209565537481857
neighbourhood_127, 0.2794539398121358
neighbourhood_71, 0.27693930856445187
neighbourhood_3, 0.26767545117618646
```

Conclusion and Next Steps:

1. While the predictability of the model was decent, it did not show any improvement with any of the regularization techniques. Further exploration can be done by using polynomial features.
2. We have 20% of the missing values in last_review and review_per_month columns. Missing value column have an impact on the price. So one improvement to our model would be to track this better.
3. Presence of outliers due to the expensive Manhattan and low priced Staten Island could have an impact on our model.
4. While most booking were done between 1 and 3 days, there are quite a few for whom the occupancy days extended beyond 100 days and for some over a year. This maybe due to customers using the listing for residence. It would help if we could separated visitors between tourist and locals.