group 11

Predicting the nightly price of AirBnB listings in New York City

1 Introduction & Context

This report outlines the process of building a model to predict the nightly price of an Airbnb listing in NYC. An open-source dataset of Aribnb listings was combined with datasets on real estate values (Zillow) and venues/places of interest (Google Places). This model could be used to recommend a suitable nightly price for a newly listed property on the Airbnb platform.

2 Predictive Analysis

Data Cleaning and Preparation

The prediction was done using three datasets, namely Listings, Venues and Real Estate.

The Listings dataset consists of 59,824 Airbnb listings along the U.S. East coast with 29 parameters. The dataset was initially filtered by metropolitan area to remove listings outside NYC. The following plot displays Airbnb price depending on location on a map of the New York metropolitan area.

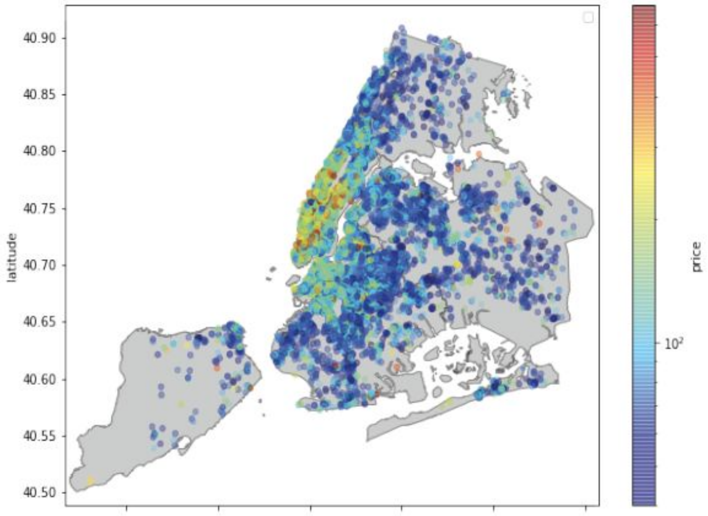


Figure 1: Airbnb listing location overlaid on NYC map with listing price on a logarithmic scale

The 6 features concerning user reviews (checkin, cleanliness, communication, location, value, rating) were then normalised to a common denominator (10). The mean value of these 6 features was then added to the dataset as a normalised_rating feature whilst dropping the six other columns.

Null/outlier data removal:

75% (23030 / 30674) of values for the weekly_price feature were missing, along with 100% of values for has_availability. As both features were considered unimportant (knowing we already have a price column), they were simply dropped. Rows with nulls in any other features were also dropped. This produced a dataset with 30089 rows and no null fields. It was decided to drop the rows instead of filling them with an imputer as the number of rows to drop was sufficiently small.

Outlier listings - defined as those for which price was more than 3 standard deviations away from the mean - were removed from the dataset. This was done in order to remove extreme values which may be fake listings or erroneous ones.

Feature Engineering:

Three additional features were added to the primary dataframe from foreign datasets. The ZRI (Zillow Rent Index) database was searched by listing zip code. If the listing zip code was present in the ZRI database, the latest ZRI value was added to the listings.

The Venues database (Google Places) was then searched by listing location. Two features (num_venues & venues_rating) were then added to the primary dataframe, adding information on the number of places of interest within a 0.01° coordinate distance (about 1.1km) of each listing, along with their average rating. The few listings without any venues in their locality were dropped.

Categorical Variable Encoding:

Non-numerical, categorical features ('room_type', 'bed_type', 'property_type', 'cancellation_policy', 'instant_bookable') were converted into sparse matrices with binary values only through Scikitlearn’s OneHotEncoder. One Hot Encoding was chosen over Label Encoding or Ordinal Encoding since the different values for each feature are nominal, with no relative weight/superiority.

Normalisation:

The final stage of preprocessing used scikitlearn’s StandardScaler to normalise all numeric features, such that each feature was rescaled to have a mean = 0 and StDev = 1. Preprocessed categorical values and the listing price were not normalized as they are respectively binary outputs and the variable we desire to predict (which should not be modified).

Classification Model:

Initially, the listings data was categorised into seven $50 price brackets. This facilitated the training and use of a classification model.

A Scikit Learn decision tree classifier was created for each price band. Each classifier was trained on a balanced subsection of the training dataset, containing equal numbers of listings in that price band and listings outside it. Since each subsection was balanced, any model that correctly predicts the binary outcome ‘is or is not in that price band’ more than 50% of the time is better than random.

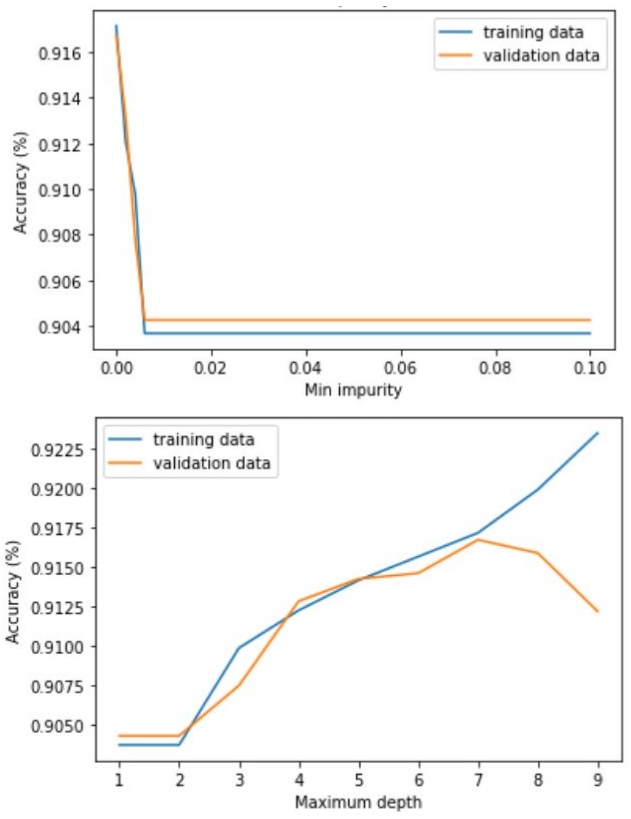


Figure 2: Min impurity and max depth for the first price band decision tree classifier

The maximum depth and minimum impurity of each tree were also plotted to ensure the model didn't overfit. A greedy algorithm was used to plot these graphs (see appendix 4.2), whereby the accuracy (an average of 1000 repeated runs) was calculated for a maximum depth up to 10. Once maximum depth was chosen, a similar algorithm was used to test the accuracy for decision trees with a minimum impurity up to 0.10.

Figure 2 shows depth and impurity data for the first decision tree. Overfitting is observed for trees with more than 7 levels (demonstrated by significant drops in the accuracy on the validation data).

The accuracies of the 7 decision trees are summarised in the table below:

Price range	**Accuracy**
p<$50	0.91
$50<p<$100	0.77
$100<p<$150	0.77
$150<p<$200	0.85
$200<p<$250	0.93
$250<p<$300	0.95
p>$300	0.95

Table 1: Decision tree accuracy for each price band

Having trained the seven decision trees and optimised the depth and impurity coefficients, predictions were then made for the test data using all seven of the trained

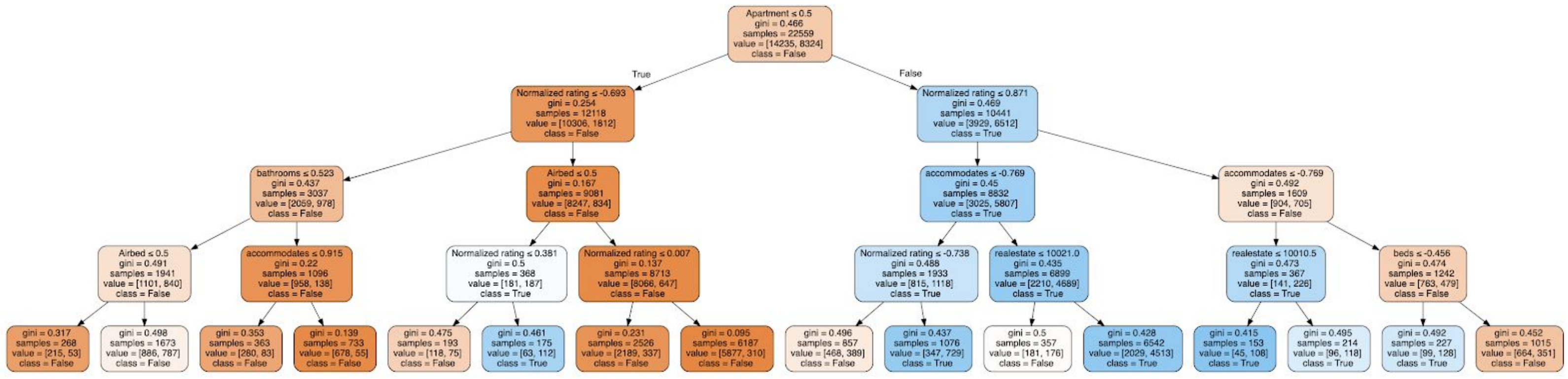

Apartment ≤ 0.5 gini = 0.466 samples = 22559 value = [14235, 8324] class = False
True
False
Normalized rating ≤ -0.693 gini = 0.254 samples = 12118 value = [10306, 1812] class = False
Normalized rating ≤ 0.871 gini = 0.469 samples = 10441 value = [3929, 6512] class = True
bathrooms ≤ 0.523 gini = 0.437 samples = 3037 value = [2059, 978] class = False
Airbed ≤ 0.5 gini = 0.167 samples = 9081 value = [8247, 834] class = False
accommodates ≤ -0.769 gini = 0.45 samples = 8832 value = [3025, 5807] class = True
accommodates ≤ -0.769 gini = 0.492 samples = 1609 value = [904, 705] class = False
Airbed ≤ 0.5 gini = 0.491 samples = 1941 value = [1101, 840] class = False
accommodates ≤ 0.915 gini = 0.22 samples = 1096 value = [958, 138] class = False
Normalized rating ≤ 0.381 gini = 0.5 samples = 368 value = [181, 187] class = True
Normalized rating ≤ 0.007 gini = 0.137 samples = 8713 value = [8066, 647] class = False
Normalized rating ≤ -0.738 gini = 0.488 samples = 1933 value = [815, 1118] class = True
realestate ≤ 10021.0 gini = 0.435 samples = 6899 value = [2210, 4689] class = True
realestate ≤ 10010.5 gini = 0.473 samples = 367 value = [141, 226] class = True
beds ≤ -0.456 gini = 0.474 samples = 1242 value = [763, 479] class = False
gini = 0.317 samples = 268 value = [215, 53] class = False
gini = 0.498 samples = 1673 value = [886, 787] class = False
gini = 0.353 samples = 363 value = [280, 83] class = False
gini = 0.139 samples = 733 value = [678, 55] class = False
gini = 0.475 samples = 193 value = [118, 75] class = False
gini = 0.461 samples = 175 value = [63, 112] class = True
gini = 0.231 samples = 2526 value = [2189, 337] class = False
gini = 0.095 samples = 6187 value = [5877, 310] class = False
gini = 0.496 samples = 857 value = [468, 389] class = False
gini = 0.437 samples = 1076 value = [347, 729] class = True
gini = 0.5 samples = 357 value = [181, 176] class = False
gini = 0.428 samples = 6542 value = [2029, 4513] class = True
gini = 0.415 samples = 153 value = [45, 108] class = True
gini = 0.495 samples = 214 value = [96, 118] class = True
gini = 0.492 samples = 227 value = [99, 128] class = True
gini = 0.452 samples = 1015 value = [664, 351] class = False

Figure 3: 1st price band decision tree classifier

trees. The final gini coefficients after classifications down the branches of the tree ranged between 0 and 0.5. The decision tree in figure 3 has mostly impure leaf nodes.

Regression model:

This predictive analysis can be seen as a regression problem as we are trying to predict an outcome based on historical data. To that extent, several initial regression models were tested at first to have an idea of which one to use. Those models included: Decision Trees, Linear Regression, K-Neighbors Regressor, Boost Gradient Descent and Random Forest Regressor. Using K-fold cross validation to split the data into 10 distinct subsets showed that the cross-validation score and standard deviation for RandomForests was better than that of the other models.

Choosing Random Forest Regressor to continue the rest of the model, we needed to find another measure of validation (as accuracy does not work for regressions). This was chosen to be MAE (median absolute error) and not RMSE as it diminishes the importance of outliers within the data. The MAE of the model before any tweaking was done was $21.2 on the validation set.

Hyperparameter tuning was deemed necessary to reduce that result. The large amount of features made us use Random Search CV instead of Grid Search CV as results are similar for a less computationally intensive process. Tuned parameters were: n_estimators, max_features, max_depth, min_samples_split, min_samples_leaf and bootstrap, Performing this on 300 instances for several parameters decreased MAE by an average of $0.57.

Following this, a final step of feature selection was performed on the fitted model. A recursive features elimination with cross validation was used to evaluate the model for different numbers of features. This method gives a score to every model and identifies the best one.

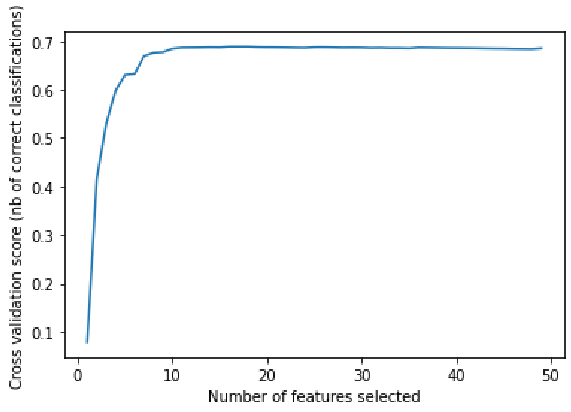


Figure 4: Cross validation score vs number of features selected graph

As we can see from the plot, the score of the model and therefore its accuracy increases up to 0.688 and plateaus past 10 features. This would indicate that any features added past the 10th one does not have an influence on the accuracy of the model and would instead overfit the model. To test this out, we attempted to create models with 10 of the top 15 ranked features identified by RFECV. However, the MAE obtained from these models was not an improvement from the last step and even worsened considerably.

By using the most optimal model defined by the RFECV, using 15 features, the model obtained had a MAE that was on average $4.46 lower than the previous model.

Model Results

Two different model categories were used on the same problem by framing it in two different manners.

Classification model:

The classification model sought to utilise binary predictive methods to categorise listings into price bands.

Had each of the seven decision trees been 100% accurate, running the test data through all of the trees would have produced one unique price-band label per listing. However, the final categorisation accuracy on test data did not exceed 87%. Consequently, it was not possible to guarantee a unique price-band prediction using this model. This is a limitation of the binary classification approach. The $50 dollar range within each band is also of limited use in the context of price recommendation to new hosts.

Regression model:

The regression model outputs a predictive Airbnb price which varies by about $16. It should be noted that results for MAE of the training set is exactly $15.86 whilst that of the validation and test set are respectively $15.95 and $15.89. This shows that the model is slightly overfitting but at percentages which can be considered negligible. When creating a $50 range around the predicted value, an accuracy of 67.5% is obtained, inferior to that obtained with the classification models.

3 Conclusion

The classification model was capable of categorising listings into $50 price bands with 87% accuracy. A substantially improved regression model was shown to have 50% of predictions within $16 of the actual price. However, if the regression model is used to predict the correct Airbnb listing price without any kind of price bracket, accuracy would drop to around 8% which shows that this model cannot perform correctly without its boundaries.

Additionally, a lot more goes into estimating the price of an Airbnb. As a matter of fact, the dataset included an ‘amenities’ column which summarized the most important facilities and features of the Airbnbs. However, each row was a dictionary of values with a different number of attributes for each listing. Transforming those into usable data was challenging and we were not able to do it which is why this column was dropped from the dataset. Better accuracies or MAE could very likely have been obtained by including this data.

Finally, it is also important to note that many other elements can influence the price of an Airbnb. Elements such as furniture, lighting and a scenic view out a window are particularly desired by Airbnb customers. As these elements are not traditionally categorised by Airbnb, it would be interesting to apply a computer vision algorithm to identify specific traits of the listings and use them as additional insight for the price estimation.

4 Appendix

4.0 Data source

https://github.com/aihack20/airbnb_challenge

4.1 Data Preprocessing

Download the data

```
# Download data - please only run this once
import urllib.request
import zipfile
import os

print("Starting download ...")
url = "https://github.com/aihack20/airbnb_challenge/releases/download/data/data-airbnb.zip"
with urllib.request.urlopen(url) as src:
    with open("tmp_airbnb_data.zip", "wb") as dest:
        dest.write(src.read())
print("Unpacking archive ...")
os.makedirs("airbnb_data", exist_ok=True)
with zipfile.ZipFile("tmp_airbnb_data.zip", compression=zipfile.ZIP_DEFLATED) as zip_f:
    zip_f.extractall("airbnb_data")
print("Done!")

import pandas as pd
from matplotlib import pyplot as plt

# Load the data (will take a short while)

listings = pd.read_csv("airbnb_data/listings.csv")
real_estate = pd.read_csv("airbnb_data/real_estate.csv")
venues = pd.read_csv("airbnb_data/venues.csv")

# remove warnings

import warnings
warnings.filterwarnings('ignore')
```

Restrict listings to ones in NYC

```
listings_NYC = listings[listings['metropolitan'] == 'NYC']

len(listings_NYC)

40740
```

Normalize and gather ranking variables

```
#normalize this score which is at 100 to get better average values

listings_NYC['review_scores_rating'] = listings_NYC['review_scores_rating']/10

#gather the differnt rating columns in a list of columns

ratings = listings_NYC.iloc[:,19:25]


list(ratings.columns)

['review_scores_checkin',
 'review_scores_cleanliness',
 'review_scores_communication',
 'review_scores_location',
 'review_scores_rating',
 'review_scores_value']


#remove rows where ratings are null

ratings_non_Na = ratings.dropna()
```

```
ratings_non_Na
```

	review_scores_checkin	review_scores_cleanliness	review_scores_communication	review_scores_location	rev
0	10.0	10.0	10.0	10.0	
3	10.0	10.0	10.0	10.0	
4	10.0	10.0	10.0	10.0	
5	10.0	10.0	10.0	10.0	
6	10.0	10.0	10.0	10.0	
...	...	...	...	...	
40727	10.0	9.0	10.0	9.0	
40730	10.0	9.0	10.0	10.0	
40731	10.0	10.0	10.0	10.0	
40732	10.0	10.0	10.0	10.0	
40736	10.0	10.0	10.0	10.0	

30674 rows × 6 columns

```
# normalize the data of the six rating columns into one

rating_avg = ratings_non_Na.sum(axis=1)/6
rating_avg
```

```
#we add the normalized data column in the Data frame, remove the other ratings column and
# update the dataframes for non-Na rows for ratings

listings_NYC['Normalized rating']=pd.Series(rating_avg)
listings_NYC= listings_NYC.drop(list(ratings),axis=1)
listings_NYC = listings_NYC[listings_NYC['Normalized rating'].notna()]
len(listings_NYC)
```

```
30674
```

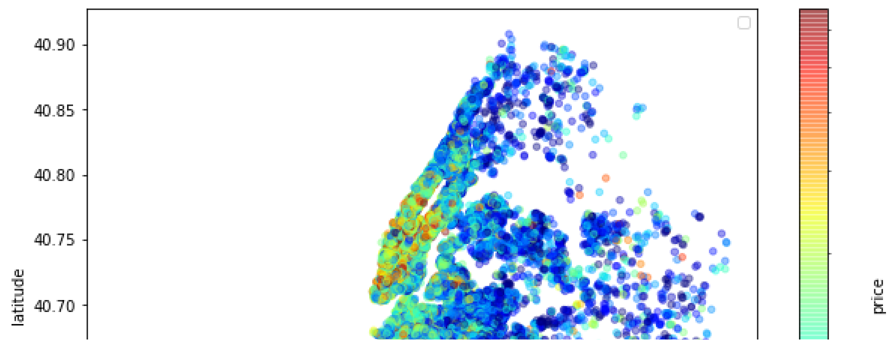
Plotting prices on a map of NYC

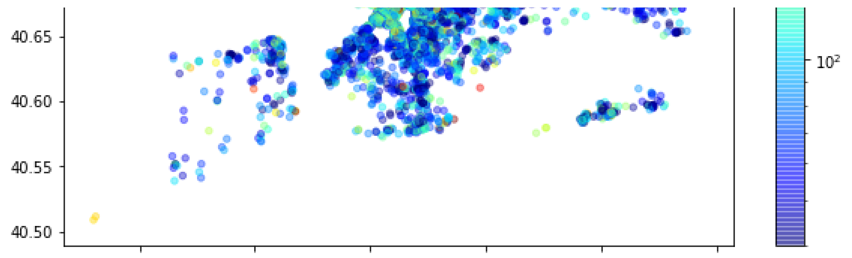
```
import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.cbook as cbook
import matplotlib.pyplot as plt

norm = mpl.colors.Normalize(vmin=-1.,vmax=1.)


listings_NYC.plot(kind='scatter', x='longitude', y='latitude', alpha=0.4,
                 figsize=(10,7), c='price', cmap=plt.get_cmap('jet'), colorbar=True,
                 norm=colors.LogNorm(vmin=listings_NYC['price'].min()*4, vmax=listings_NYC['price'].max()/15)
                 )

plt.legend()
plt.show()
```





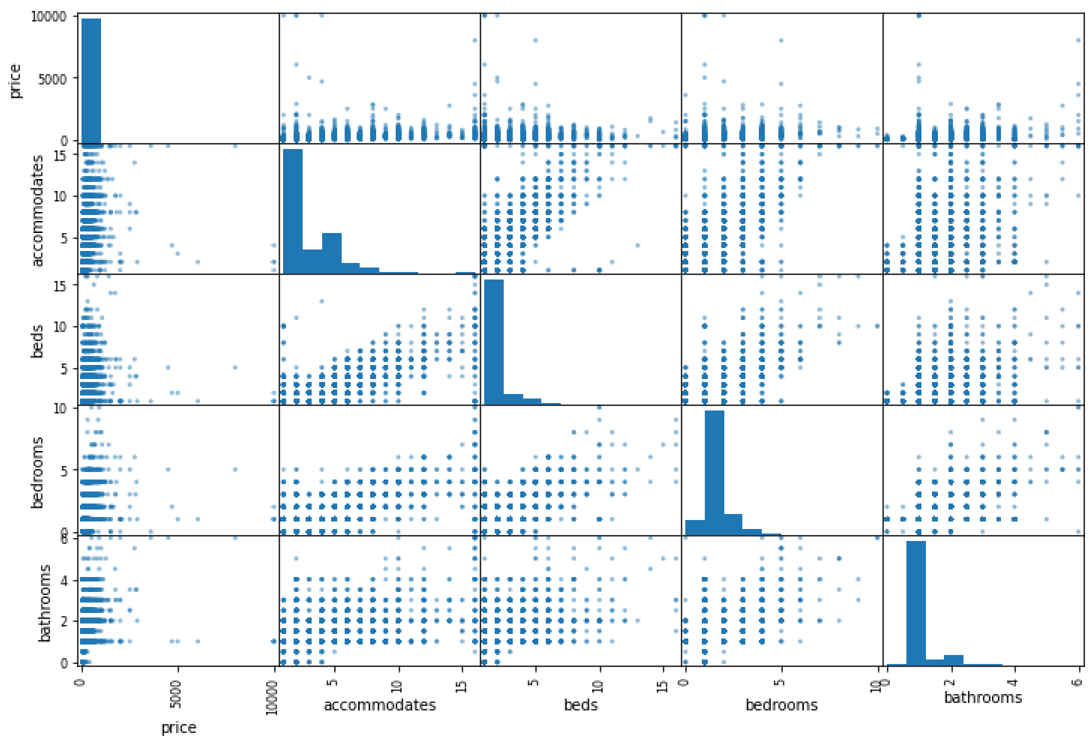
Checking for correlations

```
corr_matrix = listings_NYC.corr()
corr_matrix['price'].sort_values(ascending=False)
```

```
price                 1.000000
weekly_price          0.581100
accommodates          0.386051
beds                  0.310775
bedrooms              0.306049
bathrooms             0.262186
availability_30       0.059499
Normalized rating     0.035234
latitude              0.033099
host_id              -0.024197
id                   -0.045292
longitude            -0.171891
has_availability           NaN
Name: price, dtype: float64
```

```
from pandas.plotting import scatter_matrix

attributes = ['price','accommodates', 'beds', 'bedrooms', 'bathrooms' ]
scatter_matrix(listings_NYC[attributes], figsize=(12,8))
plt.show()
```



Getting rid of Null values and dropping columns

```
count = listings_NYC.isna().sum()
count
```

```
accommodates                0
amenities                   0
availability_30             0
bathrooms                 129
bed_type                    0
bedrooms                   57
beds                       38
cancellation_policy         0
city                       28
has_availability        30674
host_id                     0
id                          0
instant_bookable            0
latitude                    0
longitude                   0
metropolitan                0
name                       12
price                       0
property_type               0
room_type                   0
state                       0
weekly_price            23030
zipcode                   392
```

```
# According to this, there is no point in keeping 'has availability column' as well
# as the 'weekly price' as we already have a 'price' variable as well
# we also drop rows for which values for 'bedroom', 'beds' or 'bathrooms' is Na
```

```
listings_NYC.columns
```

```
count_2 = listings_NYC.isna().sum()
count_2
```

```
listings_NYC = listings_NYC.dropna(subset=['beds','bathrooms','bedrooms'])
```

```
listings_NYC = listings_NYC.dropna(subset=['zipcode'])
listings_NYC = listings_NYC.drop(['name', 'amenities','city'],axis=1)
```

```
listings_NYC = listings_NYC.drop(['host_id','availability_30'],axis=1)
```

```
count_3 = listings_NYC.isna().sum()
count_3
```

Removal of Outliers

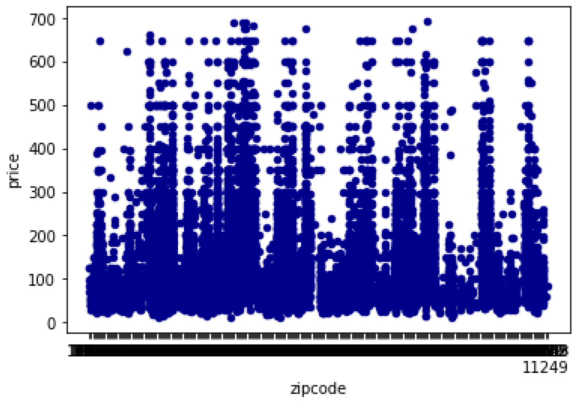
```
# Calculating the z score of every price data point, if the z score of the point
# is more than 3, then it is quite far off from the mean and the rest of the data
# will be removed from the dataset

from scipy import stats

z_scores = stats.zscore(listings_NYC['price'])

abs_z_scores = np.abs(z_scores)
filtered_entries = (abs_z_scores < 3)
listings_NYC = listings_NYC[filtered_entries]
```

```
ax1 = listings_NYC.plot.scatter(x='zipcode',y='price',c='DarkBlue')
```



Zipcode preprocessing

```
# Removing all the rows with non standard zipcode (dash, alphanumeric)
# or errors in formatting, and transform the zipcodes into a numerical
# value that can be used to plot or used more easily


listings_NYC = listings_NYC[listings_NYC.zipcode != '11103-3233']


listings_NYC = listings_NYC[listings_NYC.zipcode != '10003-8623']


listings_NYC = listings_NYC[listings_NYC.zipcode != '11385-2308']


listings_NYC = listings_NYC[listings_NYC.zipcode != '1m']


listings_NYC.loc[(listings_NYC.zipcode == '11249\n11249'),'zipcode']='11249'


listings_NYC['zipcode'] = listings_NYC['zipcode'].apply(pd.to_numeric)


# creating a list of the most recent real estate values for all of the zipcodes
# in the listing dataframe


list_zip = list(listings_NYC['zipcode'])


final_zip_list = list(set(list_zip))


real_estate_NYC = real_estate[real_estate['city']=='New York']


zip_real_estate = real_estate_NYC[['type','zipcode','2017-06']].copy()


zip_real_estate['zipcode'] = zip_real_estate['zipcode'].apply(pd.to_numeric, errors = 'coerce')


zip_real_estate = zip_real_estate[zip_real_estate['type'] != 'ZHVI']


zip_real_estate
```

	type	zipcode	2017-06
13200	ZRI	10025	3669
13202	ZRI	10023	3983
13205	ZRI	10002	3609
13207	ZRI	10467	1728
13211	ZRI	11226	2324
...	...	...	...
23669	ZRI	11363	2739
23952	ZRI	10006	3289
24257	ZRI	10004	4770
24892	ZRI	10464	1946
25197	ZRI	11005	2145

160 rows × 3 columns

Adding local real estate value to listings

```
# Adding a real estate pricing for each listing depending on their
# respective zipcode and dropping the listings with no real estate value
```

```
estate_zip = list(zip_real_estate['zipcode'])
final_estate_zip = list(set(estate_zip))
len(final_estate_zip)
```

```
160
```

```
def findEstateVal(zip):
    if zip in final_estate_zip:
        val = zip_real_estate.loc[zip_real_estate['zipcode'] == zip]['2017-06'].item()
    else:
        val = np.nan
    return val
```

```
listings_NYC['realestate'] = listings_NYC['zipcode'].apply(lambda x: findEstateVal(x))
```

```
listings_NYC = listings_NYC.dropna(axis='rows')
```

Venues Preprocessing

```
# Identifying every listed value in NYC with a rating and dropping
# every other unnecessary collumns
```

```
venues_NYC = venues[venues['city']=='new york city']
```

```
venues_NYC = venues_NYC.drop(['city','id','name','types'],axis=1)
```

```
venues_NYC = venues_NYC.dropna()
```

```
venues_NYC = venues_NYC.reset_index(drop=True)
```

Adding venues rating

```
#Adding the number of venues and average rating of the venus in a 1.1km radius
#around the airbnb
```

```
lef neighbourhood_num(lat, long):
  distance = 0.01
  venues_near = venues_NYC[((venues_NYC['latitude'] >= lat-distance)&(venues_NYC['latitude'] <=lat+distance))&((venues
  num_venues = len(venues_near)
  return num_venues
```

```
def neighbourhood_rating(lat, long):
   distance = 0.01
   venues_near = venues_NYC[((venues_NYC['latitude'] >= lat-distance)&(venues_NYC['latitude'] <=lat+distance))&((venue
   venues_rating = venues_near['rating'].mean()
   return venues_rating
```

```
listings_NYC['num_venues'] = listings_NYC.apply(lambda x: neighbourhood_num(x['latitude'], x['longitude']), axis=1)
listings_NYC['venues_rating'] = listings_NYC.apply(lambda x: neighbourhood_rating(x['latitude'], x['longitude']), axi
```

```
listings_NYC = listings_NYC.dropna()
```

Feature Engineering

```
## Instead of having two features for number of bedrooms and bathrooms, we regroup this into a rooms feature
```

```
features_df = listings_NYC.copy(deep=True)
```

```
features_df.dtypes
```

```
accommodates            float64
bathrooms               float64
bed_type                 object
bedrooms                float64
beds                    float64
cancellation_policy      object
id                        int64
instant_bookable         object
latitude                float64
longitude               float64
metropolitan             object
price                   float64
property_type            object
room_type                object
state                    object
zipcode                 float64
Normalized rating       float64
realestate              float64
num_venues                int64
venues_rating           float64
dtype: object
```

```
features_df['rooms'] = features_df['bedrooms'] + features_df['bathrooms']
```

```
features_df.drop(['bedrooms','bathrooms'],axis=1)
```

	accommodates	bed_type	beds	cancellation_policy	id	instant_bookable	latitude	longitude	metropo
0	2.0	Real Bed	1.0	moderate	7949480	f	40.852054	-73.788680	
3	3.0	Real Bed	1.0	strict	6627449	f	40.849775	-73.786609	
4	4.0	Real Bed	1.0	moderate	5557381	t	40.850024	-73.789328	
5	2.0	Real Bed	1.0	moderate	9147025	f	40.844870	-73.789541	
6	4.0	Real Bed	2.0	flexible	11675715	t	40.851391	-73.784139	
...	...	...	...	...	...	...	...	...	
40725	1.0	Real Bed	1.0	moderate	17957521	f	40.708237	-73.943789	
40726	2.0	Real Bed	1.0	strict	14942680	f	40.708192	-73.949981	
40727	5.0	Real Bed	3.0	strict	7471933	f	40.704137	-73.934081	
40730	5.0	Real Bed	2.0	moderate	6684697	f	40.708768	-73.948891	
40731	1.0	Real Bed	1.0	moderate	1835049	f	40.706738	-73.943422	

28061 rows × 19 columns

Transform categorical values with OHE

```
# to use the non-numerical data, we need to encode it without giving the different attributes different weights
# For that reason, we use OneHotEncoding instead of LabelEncoder or OrdinalEncode
```

```
from sklearn.preprocessing import OneHotEncoder
ohc = OneHotEncoder()
```

```
cat = ['room_type','bed_type', 'property_type', 'cancellation_policy','instant_bookable']
feature_cols = ohc.fit_transform(listings_NYC[cat].values.reshape(-1,1))
feature_labels = ohc.categories_[0]
feature_labels
```

```
feature_arr = ohc.fit_transform(listings_NYC[cat])
```

```
len(feature_labels)
```

```
41
```

```
fts_arr = feature_arr.toarray()
```

```
fts_arr.shape
```

```
(28061, 41)
```

```
cat_df = pd.DataFrame(data=fts_arr, columns=feature_labels)
```

```
cat_df
```

Feature correlation testing

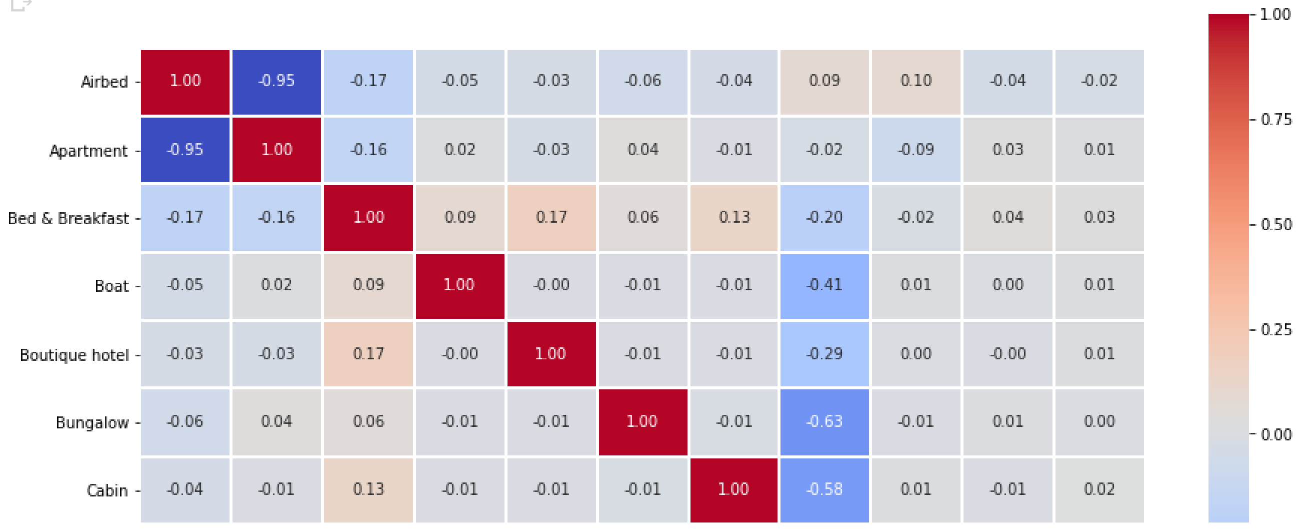
```
import seaborn as sns
```

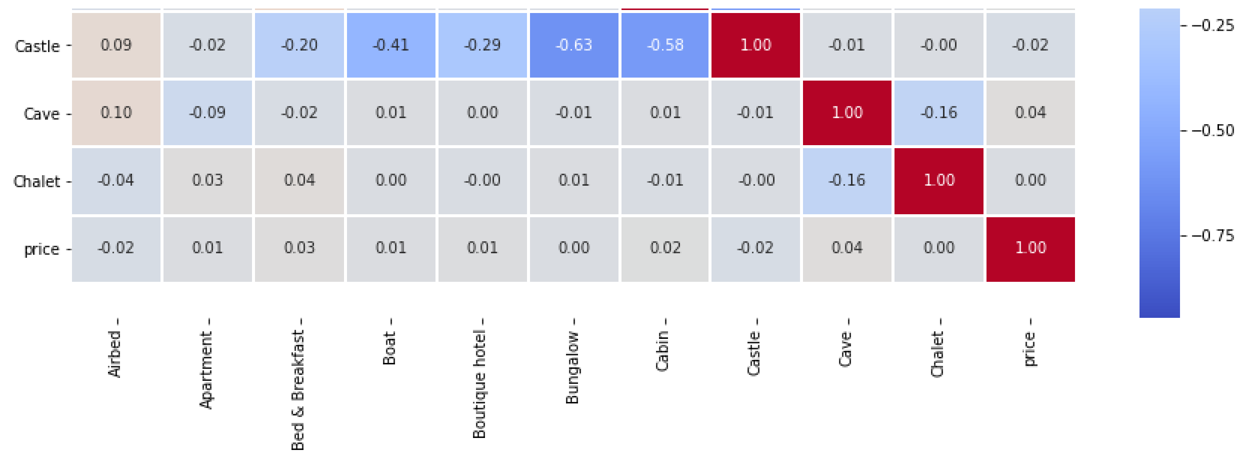
```
cols10 = cat_df.iloc[:,:10]
cols20 = cat_df.iloc[:,10:20]
cols30 = cat_df.iloc[:,20:30]
cols40 = cat_df.iloc[:,30:]
```

```
cols10['price'] = listings_NYC['price']
cols20['price'] = listings_NYC['price']
cols30['price'] = listings_NYC['price']
cols40['price'] = listings_NYC['price']
```

```
# we do the correlation matrix as something more visually appealing
# seems to shows more negative correlation than positive one
# correlations with price are very little however

corr_matrix = cols10.corr()
fig, ax = plt.subplots(figsize=(15,10))
ax = sns.heatmap(corr_matrix, annot=True, linewidths=1,
                fmt=".2f",cmap="coolwarm")
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top -0.5)
```





Normalizing the preprocessed data

```
# we normalize the data before using it. Using StandardScaler means the data will have
# a mean of 0 with a standard deviation of 1

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
num_values = listings_NYC.select_dtypes(include=numerics)
num_values_2 = num_values.copy(deep=True)


num_values = num_values.drop(['id','latitude','longitude','zipcode','price'],axis=1)


num_values
```

	accommodates	bathrooms	bedrooms	beds	Normalized rating	realestate	num_venues	venues_rating
0	2.0	1.0	1.0	1.0	10.000000	1946.0	51	4.345098
3	3.0	1.0	1.0	1.0	9.883333	1946.0	54	4.325926
4	4.0	1.0	1.0	1.0	9.950000	1946.0	54	4.325926
5	2.0	1.0	0.0	1.0	9.950000	1946.0	60	4.296667
6	4.0	1.0	1.0	2.0	9.966667	1946.0	51	4.345098
...	...	...	...	...	...	...	...	...
40725	1.0	1.0	1.0	1.0	9.666667	2702.0	879	4.186007
40726	2.0	1.0	1.0	1.0	9.000000	2702.0	1225	4.246857
40727	5.0	1.0	3.0	3.0	9.633333	2702.0	816	4.131618
40730	5.0	1.0	1.0	2.0	9.600000	2702.0	1157	4.229818
40731	1.0	1.5	4.0	1.0	9.933333	2702.0	880	4.150341

28061 rows × 8 columns

```
scaler.fit(num_values)
scaler.mean_
```

```
array([2.87060333e+00, 1.11685257e+00, 1.14447097e+00, 1.55981611e+00,
       9.45259138e+00, 3.17081180e+03, 1.22454389e+03, 4.11119578e+00])
```

```
num_scaled = scaler.fit_transform(num_values)
print(num_scaled.shape)
```

```
(28061, 8)
```

```
num_scaled
```

```
array([[-0.48871648, -0.32436157, -0.21264662, ..., -1.63048868,
        -1.17356607,  2.19316508],
       [ 0.07263731, -0.32436157, -0.21264662, ..., -1.63048868,
        -1.17056602,  2.01339935],
       [ 0.6339911 , -0.32436157, -0.21264662, ..., -1.63048868,
        -1.17056602,  2.01339935],
       ...,
       [ 1.19534489, -0.32436157,  2.73115049, ..., -0.62408963,
        -0.40855161,  0.19148396],
       [ 1.19534489, -0.32436157, -0.21264662, ..., -0.62408963,

       [-1.05007026,  1.06354784,  4.20304904, ..., -0.62408963,
        -0.3445504 ,  0.36704107]])
```

```
ccols = list(num_values.columns)
```

```
num_scaled = pd.DataFrame(num_scaled, columns=cols)
```

```
num_values_2=num_values_2.reset_index(drop=True)
```

```
num_values_2
```

	accommodates	bathrooms	bedrooms	beds	id	latitude	longitude	price	zipcode	Normalized rating	realesta
0	2.0	1.0	1.0	1.0	7949480	40.852054	-73.788680	99.0	10464.0	10.000000	194
1	3.0	1.0	1.0	1.0	6627449	40.849775	-73.786609	125.0	10464.0	9.883333	194
2	4.0	1.0	1.0	1.0	5557381	40.850024	-73.789328	69.0	10464.0	9.950000	194
3	2.0	1.0	0.0	1.0	9147025	40.844870	-73.789541	125.0	10464.0	9.950000	194
4	4.0	1.0	1.0	2.0	11675715	40.851391	-73.784139	85.0	10464.0	9.966667	194
...	...	...	...	...	...	...	...	...	...	...	
28056	1.0	1.0	1.0	1.0	17957521	40.708237	-73.943789	59.0	11206.0	9.666667	270
28057	2.0	1.0	1.0	1.0	14942680	40.708192	-73.949981	90.0	11206.0	9.000000	270
28058	5.0	1.0	3.0	3.0	7471933	40.704137	-73.934081	250.0	11206.0	9.633333	270
28059	5.0	1.0	1.0	2.0	6684697	40.708768	-73.948891	125.0	11206.0	9.600000	270
28060	1.0	1.5	4.0	1.0	1835049	40.706738	-73.943422	59.0	11206.0	9.933333	270

28061 rows × 13 columns

Final dataframe to use

```
num_df = num_scaled
num_df['price'] = num_values_2['price']
```

```
df = num_df.join(cat_df)
```

Splitting the data

```
from sklearn.model_selection import train_test_split

y = df['price']
X = df.drop(['price'],axis=1)

train, other = train_test_split(df, test_size = 0.2, random_state = 42)
validation, test = train_test_split(other, test_size = 0.5, random_state=42)
```

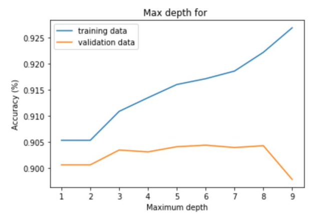
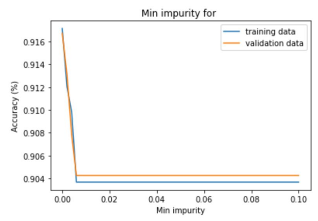
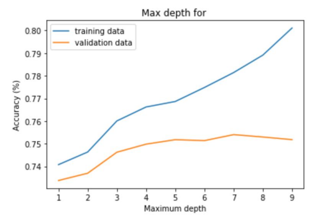
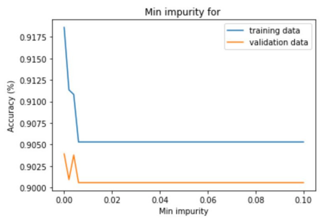
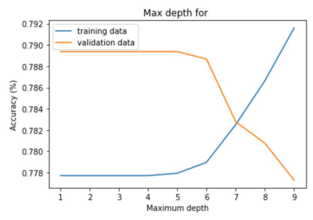
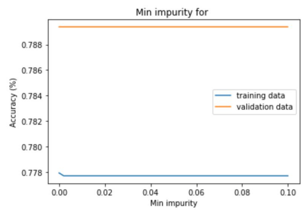
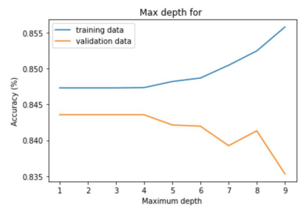
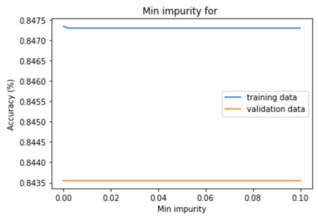
```
X_train = train.drop(columns=['price'])
y_train = train['price']
X_validation = validation.drop(columns=['price'])
y_validation = validation['price']
X_test = test.drop(columns=['price'])
y_test = test['price']
```

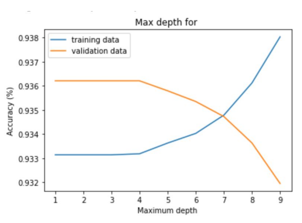
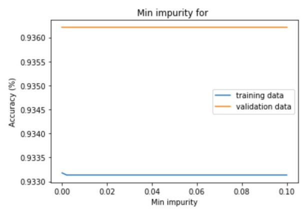
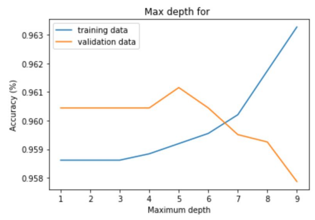
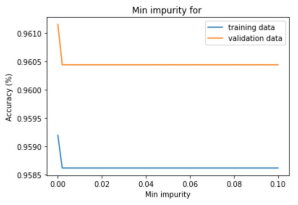
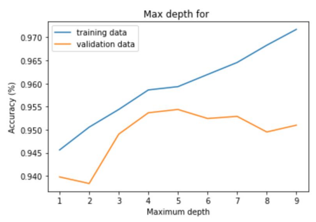
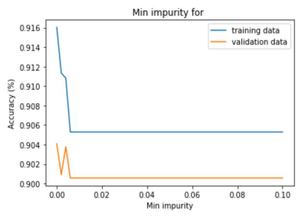
4.2 Classification model

Forward selection

Data set	Selected columns	Accuracy
p<50	['accommodates', 'realestate', 'beds']	0.90
50<p<100	['Apartment', 'realestate', 'accommodates', 'super_strict_30']	0.79
100<p<150	['accommodates']	0.79
150<p<200	['bathrooms']	0.84
200<p<250	['bathrooms']	0.94
250<p<300	['bathrooms']	0.96
p>300	['bedrooms', 'realestate', 'accommodates', 'bathrooms']	0.95

Separate decision trees

Price range	Max depth graph	Min impurity graph	Chosen values
p < 50			Max depth - 8 Minimum impurity - 0
50 < p < 100			Max depth - 7 Minimum impurity - 0
100<p<150			Max depth - 5 Minimum impurity - 0
150<p<200			Max depth - 4 Minimum impurity - 0

200<p<250			Max depth - 4 Minimum impurity - 0
250<p<300			Max depth - 5 Minimum impurity - 0
p> 300			Max depth - 5 Minimum impurity - 0

Decision tree method:
Splitting data into test, train and validation sets

```
columns_to_add = ['price_range_0', 'price_range_1',
'price_range_2','price_range_3','price_range_4','price_range_5', 'price_range_6']

all_columns = [ [] for _ in range(7) ]
X_train = [ [] for _ in range(7) ]
y_train = [ [] for _ in range(7) ]
X_test = [ [] for _ in range(7) ]
y_test = [ [] for _ in range(7) ]
X_validation = [ [] for _ in range(7) ]
y_validation = [ [] for _ in range(7) ]

for i in range(len(columns_to_add)):
       X_train[i] = train.drop(columns=['price_range_0', 'price_range_1', 'price_range_2',
'price_range_3', 'price_range_4','price_range_5', 'price_range_6', 'price'])
       y_train[i] = train[columns_to_add[i]].values.reshape(-1,1)
       X_test[i] = test.drop(columns=['price_range_0', 'price_range_1',
'price_range_2','price_range_3','price_range_4','price_range_5', 'price_range_6', 'price'])
       y_test[i] = test[columns_to_add[i]].values.reshape(-1,1)
       X_validation[i] = validation.drop(columns=['price_range_0', 'price_range_1',
'price_range_2','price_range_3','price_range_4','price_range_5', 'price_range_6', 'price'])
       y_validation[i] = validation[columns_to_add[i]].values.reshape(-1,1)

       clf[i] = DecisionTreeClassifier(max_depth=7, min_impurity_decrease = 0.03)
       clf[i].fit(X_train[i],y_train[i])
```

Maximum depth (repeated for all prices)

```
i = 0
x_list = list(range(1,10))
train_list = []
val_list = []
count = list(range(1,1001))
number = 0

for x in x_list:
 temptrain = []
 temp = []
 for y in count:
   model = tree.DecisionTreeClassifier(max_depth=x, min_impurity_decrease = 0)
   model.fit(X_train[i],y_train[i])
   y_pred = model.predict(X_validation[i])
   y_pred_train = model.predict(X_train[i])
   acc_train = accuracy_score(y_train[i], y_pred_train)
   acc = accuracy_score(y_validation[i],y_pred)
   prec = precision_score(y_validation[i],y_pred,
                          pos_label='positive',
                          average='micro')
   rec = recall_score(y_validation[i],y_pred,
                      pos_label='positive',
                      average='micro')
   temptrain.append(acc_train)
   temp.append(acc)
   if y == 1000:
     meantrain = sum(temptrain)/len(temptrain)
     meanval = sum(temp)/len(temp)
     train_list.append(meantrain)
     val_list.append(meanval)
     number = number + 1
     print(number)

plot1, = plt.plot(x_list,train_list)
plot2, = plt.plot(x_list,val_list)
plt.title('Max depth for ')
plt.ylabel('Accuracy (%)')
plt.xlabel('Maximum depth')
plt.legend([plot1, plot2], ['training data','validation data'])
```

Minimum impurity decrease (repeated for all prices)

```
x_list =  np.linspace(0,0.1,51)
train_list= []
val_list = []
count = list(range(1,101))
number = 0

for x in x_list:
 temptrain = []
 temp = []
 for y in count:
   model = tree.DecisionTreeClassifier(max_depth=8, min_impurity_decrease = x)
   model.fit(X_train[i],y_train[i])
   y_pred = model.predict(X_validation[i])
   y_pred_train = model.predict(X_train[i])
   acc_train = accuracy_score(y_train[i],y_pred_train)
   acc = accuracy_score(y_validation[i],y_pred)
   prec = precision_score(y_validation[i],y_pred)
   rec = recall_score(y_validation[i],y_pred)
   temptrain.append(acc_train)
   temp.append(acc)
   if y == 100:
     meantrain = sum(temptrain)/len(temptrain)
     meanval = sum(temp)/len(temp)
     train_list.append(meantrain)
     val_list.append(meanval)
     number = number + 1
     print(number)

plot1, = plt.plot(x_list,train_list)
plot2, = plt.plot(x_list,val_list)
plt.title('Min impurity for ')
plt.ylabel('Accuracy (%)')
plt.xlabel('Min impurity')
plt.legend([plot1, plot2], ['training data','validation data'])
```

Building and displaying decision tree (repeated for all prices)

```
clf0 = DecisionTreeClassifier(max_depth=8, min_impurity_decrease = 0)
clf0.fit(X_train[i],y_train[i])
print('Accuracy on the val set: ',accuracy_score(y_validation[i],
clf0.predict(X_validation[i])))

dot_data = tree.export_graphviz(clf0, out_file=None)
graph = graphviz.Source(dot_data)
predictors = X_train[i].columns
dot_data = tree.export_graphviz(clf0, out_file=None, feature_names = clf0['feature_names'],
                               class_names = ('False', 'True'),
                               filled = True, rounded = True,
                               special_characters = True)
graph = graphviz.Source(dot_data)
graph
```

Test set model predictions:

```
clf_list = [clf0, clf1, clf2, clf3, clf4, clf5, clf6]
y_preds = [ [] for _ in range(7) ]
acc_list = []

for i in range(len(clf_list)):
 y_preds[i] = clf_list[i].predict(X_test)
 acc_list.append(accuracy_score(y_test[i], y_preds[i]))

mean_acc = sum(acc_list)/len(acc_list)
print(mean_acc)
```

4.3 Regression model

Training regression model

```
# Different basic regression models are tested to see how each one performs


# creating a basic decision tree

from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(X_train,y_train)


# we use sklearn's K-fold cross validation feature to split the training set in 10 distinct subsets
# on which the Decision Tree model can work on

from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg,X_train,y_train,
                         scoring='neg_mean_squared_error', cv=10)
tree_rmse_scores = np.sqrt(-scores)


#let us check initial results

def display_scores(scores):
  print('Scores:', scores)
  print('Mean:', scores.mean())
  print('Standard Deviation:', scores.std())

display_scores(tree_rmse_scores)


# lets see how linear regression works here
# seems like linear regression performs better

from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()

lin_scores = cross_val_score(lin_reg, X_train, y_train,
                             scoring='neg_mean_squared_error', cv=10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)


# Gradient boosting regressor gives us a very good result
# no improvement with ramdom search

from sklearn.ensemble import GradientBoostingRegressor

reg = GradientBoostingRegressor(random_state=42)
reg.fit(X_train, y_train)

ypred = reg.predict(X_validation)

two_features_mse = mean_squared_error(y_validation, ypred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_validation, ypred)

print('Validation set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))

ypred = reg.predict(X_test)

two_features_mse = mean_squared_error(y_test, ypred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_test, ypred)

print('\nTest set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))
```

```
Validation set:
root mean squared error: 49.596739471547394
mean squared error: 2459.8365662085475
median absolute error: 20.961793463208295

Test set:
root mean squared error: 46.633023295860305
mean squared error: 2174.6388617122498
median absolute error: 20.816714124266852
```

```
from sklearn.neighbors import KNeighborsRegressor

neigh = KNeighborsRegressor()
neigh.fit(X_train, y_train)

ypred = neigh.predict(X_validation)

two_features_mse = mean_squared_error(y_validation, ypred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_validation, ypred)

print('Validation set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))

ypred = neigh.predict(X_test)

two_features_mse = mean_squared_error(y_test, ypred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_test, ypred)

print('\nTest set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))
```

```
Validation set:
root mean squared error: 53.68168316685417
mean squared error: 2881.723107626515
median absolute error: 22.800000000000004

Test set:
root mean squared error: 52.0613674226135
mean squared error: 2710.3859779123623
median absolute error: 22.799999999999997
```

```
# # bad overfitting of the DecisionTree, we decide to use a RandomForestRegressor
# # seems to work much better so we continue with this model

from sklearn.ensemble import RandomForestRegressor
forest_reg = RandomForestRegressor()
forest_scores = cross_val_score(forest_reg, X_train, y_train,
                             scoring='neg_mean_squared_error', cv=10)
model = forest_reg.fit(X_train,y_train)
forest_rmse = np.sqrt(-forest_scores)
display_scores(forest_rmse)
```

Random Forest Regressor Training

```
import sklearn as sk
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import accuracy_score, precision_score, recall_score


model = RandomForestRegressor(random_state=42)
model = model.fit(X_train,y_train)

ypred = model.predict(X_validation)
```

```
from sklearn.metrics import median_absolute_error
from sklearn.metrics import mean_squared_error


testpred = model.predict(X_validation)

two_features_mse = mean_squared_error(y_validation, testpred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_validation, testpred)

print('Validation set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))

testpred = model.predict(X_test)

two_features_mse = mean_squared_error(y_test, testpred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_test, testpred)

print('\nTest set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))
```

Hyperparameter Tuning

```
# RandomizedSearchCV creates all the possible combinations of the parameters for
# the random forest given and tries each one to identify the most accurate


# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}


from sklearn.model_selection import RandomizedSearchCV

model = RandomForestRegressor()
model_random = RandomizedSearchCV(estimator = model, param_distributions = random_grid, n_iter = 100, cv = 3, verbose


model_random = model_random.fit(X_train,y_train)
```

```
model_random.best_params_
```

```
testpred = model.predict(X_validation)

two_features_mse = mean_squared_error(y_validation, testpred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_validation, testpred)

print('Validation set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))

testpred = model.predict(X_test)

two_features_mse = mean_squared_error(y_test, testpred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_test, testpred)

print('\nTest set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))
```

Recursive Feature Elimination with Cross-Validation

```
# Feature ranking with recursive feature elimination and cross-validated
# selection of the best number of features.
```

```
from sklearn.feature_selection import RFECV
```

```
selector = RFECV(model, step=1, cv=3, verbose=2)
selector = selector.fit(X_train,y_train)
```

```
testpred = selector.predict(X_validation)

two_features_mse = mean_squared_error(y_validation, testpred)
two_features_rmse = two_features_mse ** (1/2)

med_error = median_absolute_error(y_validation, testpred)

print('Validation set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))

testpred = selector.predict(X_test)

two features mse = mean squared error(y test  testpred)
two_features_mse = mean_squared_error(y_test, testpred)
two_features_rmse = two_features_mse ** (1/2)
```

```
med_error = median_absolute_error(y_test, testpred)

print('\nTest set:')
print('root mean squared error: {}'.format(two_features_rmse))
print('mean squared error: {}'.format(two_features_mse))
print('median absolute error: {}'.format(med_error))
```

```
Validation set:
root mean squared error: 38.299738764678345
mean squared error: 1466.8699894426054
median absolute error: 15.952864862751632

Test set:
root mean squared error: 36.33182915440095
mean squared error: 1320.0018097045786
median absolute error: 15.888422800414418
```

```
ranks = list(selector.ranking_)
```

```
selector.get_params()
```

```
{'cv': 3,
 'estimator': RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                       max_depth=20, max_features='sqrt', max_leaf_nodes=None,
                       max_samples=None, min_impurity_decrease=0.0,
                       min_impurity_split=None, min_samples_leaf=1,
                       min_samples_split=10, min_weight_fraction_leaf=0.0,
                       n_estimators=1600, n_jobs=None, oob_score=False,
                       random_state=42, verbose=0, warm_start=False),
 'estimator__bootstrap': True,
 'estimator__ccp_alpha': 0.0,
 'estimator__criterion': 'mse',
 'estimator__max_depth': 20,
 'estimator__max_features': 'sqrt',
 'estimator__max_leaf_nodes': None,
 'estimator__max_samples': None,
 'estimator__min_impurity_decrease': 0.0,
 'estimator__min_impurity_split': None,
 'estimator__min_samples_leaf': 1,
 'estimator__min_samples_split': 10,
 'estimator__min_weight_fraction_leaf': 0.0,
 'estimator__n_estimators': 1600,
 'estimator__n_jobs': None,
 'estimator__oob_score': False,
 'estimator__random_state': 42,
 'estimator__verbose': 0,
 'estimator__warm_start': False,
 'min_features_to_select': 1,
 'n_jobs': None,
 'scoring': None,
 'step': 1,
 'verbose': 2}
```

```
# Plotting the score vs selected features graph to see where overfitting begins

plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Cross validation score (nb of correct classifications)")
plt.plot(range(1, len(selector.grid_scores_) + 1), selector.grid_scores_)
plt.show()
```

```
# Output all of the sorted parameters by optimal ranking identified by RFECV

all_cols =list(X_train.columns)
dict_rank = {'Parameter':all_cols,'Ranking':ranks}
rank_df = pd.DataFrame(dict_rank)
rank_df = rank_df.sort_values('Ranking')
rank_df = rank_df.reset_index(drop=True)
```

```
rank_df[:17]
```

	Parameter	Ranking
0	accommodates	1
1	bathrooms	1
2	realestate	1
3	bedrooms	1
4	Normalized rating	1
5	Airbed	1
6	venues_rating	1
7	beds	1
8	t	1
9	Bed & Breakfast	1
10	Cave	1
11	Guesthouse	1
12	Real Bed	1
13	f	1
14	no_refunds	1
15	Apartment	1
16	num_venues	1

Converting exact price to price range

```
predict_copy = ypred.copy()


test = np.ndarray(shape=(len(predict_copy),2), dtype=float, buffer = np.zeros(2*len(predict_copy)))


for i in range(len(predict_copy)):
    test[i]= [predict_copy[i]-25,predict_copy[i]+25]
test
```

```
array([[ 60.26733738, 110.26733738],
       [ 58.88963319, 108.88963319],
       [ 29.39413867,  79.39413867],
       ...,
       [ 94.07094445, 144.07094445],
       [ 27.22633709,  77.22633709],
       [ 37.34260714,  87.34260714]])
```

```
correct = 0

list_correct = list(y_validation)

for i in range(len(predict_copy)):
  if test[i][0]<=list_correct[i]<=test[i][1]:
    correct += 1


correct/len(predict_copy)
```

```
0.6752182433636201
```