|  |
| --- |
| In this post we’re going to model the prices of Airbnb appartments in London. In other words, the aim is to build our own price suggestion model. We will be using data from <http://insideairbnb.com/> which we collected in April 2018. This work is inspired from the [Airbnb price prediction model](https://d1no007.github.io/OptiBnB/)built by Dino Rodriguez, Chase Davis, and Ayomide Opeyemi. Normally we would be doing this in R but we thought we’d try our hand at Python for a change.  **Data Preprocessing**  First, we import the listings gathered in the csv file.  import pandas as pd  listings\_file\_path = 'listings.csv.gz'  listings = pd.read\_csv(listings\_file\_path, compression="gzip", low\_memory=False)  listings.columns  Index(['id', 'listing\_url', 'scrape\_id', 'last\_scraped', 'name', 'summary',  'space', 'description', 'experiences\_offered', 'neighborhood\_overview',  'notes', 'transit', 'access', 'interaction', 'house\_rules',  'thumbnail\_url', 'medium\_url', 'picture\_url', 'xl\_picture\_url',  'host\_id', 'host\_url', 'host\_name', 'host\_since', 'host\_location',  'host\_about', 'host\_response\_time', 'host\_response\_rate',  'host\_acceptance\_rate', 'host\_is\_superhost', 'host\_thumbnail\_url',  'host\_picture\_url', 'host\_neighbourhood', 'host\_listings\_count',  'host\_total\_listings\_count', 'host\_verifications',  'host\_has\_profile\_pic', 'host\_identity\_verified', 'street',  'neighbourhood', 'neighbourhood\_cleansed',  'neighbourhood\_group\_cleansed', 'city', 'state', 'zipcode', 'market',  'smart\_location', 'country\_code', 'country', 'latitude', 'longitude',  'is\_location\_exact', 'property\_type', 'room\_type', 'accommodates',  'bathrooms', 'bedrooms', 'beds', 'bed\_type', 'amenities', 'square\_feet',  'price', 'weekly\_price', 'monthly\_price', 'security\_deposit',  'cleaning\_fee', 'guests\_included', 'extra\_people', 'minimum\_nights',  'maximum\_nights', 'calendar\_updated', 'has\_availability',  'availability\_30', 'availability\_60', 'availability\_90',  'availability\_365', 'calendar\_last\_scraped', 'number\_of\_reviews',  'first\_review', 'last\_review', 'review\_scores\_rating',  'review\_scores\_accuracy', 'review\_scores\_cleanliness',  'review\_scores\_checkin', 'review\_scores\_communication',  'review\_scores\_location', 'review\_scores\_value', 'requires\_license',  'license', 'jurisdiction\_names', 'instant\_bookable',  'cancellation\_policy', 'require\_guest\_profile\_picture',  'require\_guest\_phone\_verification', 'calculated\_host\_listings\_count',  'reviews\_per\_month'],  dtype='object')  The data has 95 columns or features. Our first step is to perform feature selection to reduce this number.  **Feature selection**  **Selection on Missing Data**  Features that have a high number of missing values aren’t useful for our model so we should remove them.  import matplotlib.pyplot as plt  %matplotlib inline  percentage\_missing\_data = listings.isnull().sum() / listings.shape[0]  ax = percentage\_missing\_data.plot(kind = 'bar', color='#E35A5C', figsize = (16, 5))  ax.set\_xlabel('Feature')  ax.set\_ylabel('Percent Empty / NaN')  ax.set\_title('Feature Emptiness')  plt.show()  As we can see, the features neighbourhood\_group\_cleansed, square\_feet, has\_availability, license and jurisdiction\_names mostly have missing values. The features neighbourhood, cleaning\_fee and security\_deposit are more than 30% empty which is too much in our opinion. The zipcode feature also has some missing values but we can either remove these values or impute them within reasonable accuracy.  useless = ['neighbourhood', 'neighbourhood\_group\_cleansed', 'square\_feet', 'security\_deposit', 'cleaning\_fee',  'has\_availability', 'license', 'jurisdiction\_names']  listings.drop(useless, axis=1, inplace=True)  **Selection on Sparse Categorical Features**  Let’s have a look at the categorical data to see the number of unique values.  categories = listings.columns[listings.dtypes == 'object']  percentage\_unique = listings[categories].nunique() / listings.shape[0]  ax = percentage\_unique.plot(kind = 'bar', color='#E35A5C', figsize = (16, 5))  ax.set\_xlabel('Feature')  ax.set\_ylabel('Percent # Unique')  ax.set\_title('Feature Emptiness')  plt.show()  We can see that the street and amenities features have a large number of unique values. It would require some natural language processing to properly wrangle these into useful features. We believe we have enough location information with neighbourhood\_cleansed and zipcode so we’ll remove street. We also remove amenities, calendar\_updated and calendar\_last\_updated features as these are too complicated to process for the moment.  to\_drop = ['street', 'amenities', 'calendar\_last\_scraped', 'calendar\_updated']  listings.drop(to\_drop, axis=1, inplace=True)  Now, let’s have a look at the zipcode feature. The above visualisation shows us that there are lots of different postcodes, maybe too many?  print("Number of Zipcodes:", listings['zipcode'].nunique())  Number of Zipcodes: 24774  Indeed, there are too many zipcodes. If we leave this feature as is it might cause overfitting. Instead, we can regroup the postcodes. At the moment, they are separated as in the following example: KT1 1PE. We’ll keep the first part of the zipcode (e.g. KT1) and accept that this gives us some less precise location information.  listings['zipcode'] = listings['zipcode'].str.slice(0,3)  listings['zipcode'] = listings['zipcode'].fillna("OTHER")  print("Number of Zipcodes:", listings['zipcode'].nunique())  Number of Zipcodes: 461  A lot of zipcodes contain less than 100 apartments and a few zipcodes contain most of the apartments. Let’s keep these ones.  relevant\_zipcodes = count\_per\_zipcode[count\_per\_zipcode > 100].index  listings\_zip\_filtered = listings[listings['zipcode'].isin(relevant\_zipcodes)]  # Plot new zipcodes distribution  count\_per\_zipcode = listings\_zip\_filtered['zipcode'].value\_counts()  ax = count\_per\_zipcode.plot(kind='bar', figsize = (22,4), color = '#E35A5C', alpha = 0.85)  ax.set\_title("Zipcodes by Number of Listings")  ax.set\_xlabel("Zipcode")  ax.set\_ylabel("# of Listings")  plt.show()  print('Number of entries removed: ', listings.shape[0] - listings\_zip\_filtered.shape[0])  Number of entries removed: 5484  This distribution is much better, and we only removed 5484 rows from our dataframe which contained about 53904 rows.  **Selection on Correlated Features**  Next we look at correlations.  import numpy as np  from sklearn import preprocessing  # Function to label encode categorical variables.  # Input: array (array of values)  # Output: array (array of encoded values)  def encode\_categorical(array):  if not array.dtype == np.dtype('float64'):  return preprocessing.LabelEncoder().fit\_transform(array)  else:  return array    # Temporary dataframe  temp\_data = listings\_neighborhood\_filtered.copy()  # Delete additional entries with NaN values  temp\_data = temp\_data.dropna(axis=0)  # Encode categorical data  temp\_data = temp\_data.apply(encode\_categorical)  # Compute matrix of correlation coefficients  corr\_matrix = temp\_data.corr()  # Display heat map  plt.figure(figsize=(7, 7))  plt.pcolor(corr\_matrix, cmap='RdBu')  plt.xlabel('Predictor Index')  plt.ylabel('Predictor Index')  plt.title('Heatmap of Correlation Matrix')  plt.colorbar()  plt.show()  This reveals that calculated\_host\_listings\_count is highly correlated with host\_total\_listings\_count so we’ll keep the latter. We also see that the availability\_\* variables are correlated with each other. We’ll keep availability\_365 as this one is less correlated with other variables. Finally, we decide to drop requires\_license which has an odd correlation result of NA’s which will not be useful in our model.  useless = ['calculated\_host\_listings\_count', 'availability\_30', 'availability\_60', 'availability\_90', 'requires\_license']  listings\_processed = listings\_neighborhood\_filtered.drop(useless, axis=1) |

**Data Splitting: Features / labels – Training set / testing set**

Now we split into features and labels and training and testing sets. We also convert the train and test dataframe into numpy arrays so that they can be used to train and test the models.

# Shuffle the data to ensure a good distribution for the training and testing sets

from sklearn.utils import shuffle

listings\_processed = shuffle(listings\_processed)

# Extract features and labels

y = listings\_processed['price']

X = listings\_processed.drop('price', axis = 1)

# Training and Testing Sets

from sklearn.model\_selection import train\_test\_split

train\_X, test\_X, train\_y, test\_y = train\_test\_split(X, y, random\_state = 0)

train\_X = np.array(train\_X)

test\_X = np.array(test\_X)

train\_y = np.array(train\_y)

test\_y = np.array(test\_y)

train\_X.shape, test\_X.shape

((36185, 170), (12062, 170))

**Modelling**

Now that the data preprocessing is over, we can start the second part of this work: applying different Machine Learning models. We decided to apply 3 different models:

* Random Forest, with the RandomForestRegressor from the Scikit-learn library
* Gradient Boosting method, with the XGBRegressor from the XGBoost library
* Neural Network, with the MLPRegressor from the Scikit-learn library.

Each time, we applied the model with its default hyperparameters and we then tuned the model in order to get the best hyperparameters. The metrics we use to evaluate the models are the median absolute error due to the presence of extreme outliers and skewness in the data set.

**Application of the Random Forest Regressor**

Let’s start with the Random Forest model.

**With default hyperparameters**

We first create a pipeline that imputes the missing values then scales the data and finally applies the model. We then fit this pipeline to the training set.

from sklearn.ensemble import RandomForestRegressor

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import Imputer

from sklearn.preprocessing import StandardScaler

# Create the pipeline (imputer + scaler + regressor)

my\_pipeline\_RF = make\_pipeline(Imputer(), StandardScaler(),

RandomForestRegressor(random\_state=42))

# Fit the model

my\_pipeline\_RF.fit(train\_X, train\_y)

We evaluate this model on the test set, using the median absolute error to measure the performance of the model. We’ll also include the root-mean-square error (RMSE) for completeness. Since we’ll be doing this repeatedly it is good practice to create a function.

from sklearn.metrics import median\_absolute\_error

from sklearn.metrics import mean\_squared\_error

from math import sqrt

def evaluate\_model(model, predict\_set, evaluate\_set):

predictions = model.predict(predict\_set)

print("Median Absolute Error: " + str(round(median\_absolute\_error(predictions, evaluate\_set), 2)))

RMSE = round(sqrt(mean\_squared\_error(predictions, evaluate\_set)), 2)

print("RMSE: " + str(RMSE))

evaluate\_model(my\_pipeline\_RF, test\_X, test\_y)

Median Absolute Error: 14.2

RMSE: 126.16

**Hyperparameters tuning**

We had some good results with the default hyperparameters of the Random Forest regressor. But we can improve the results with some hyperparameter tuning. There are two main methods available for this:

* Random search
* Grid search

You have to provide a parameter grid to these methods. Then, they both try different combinations of parameters within the grid you provided. But the first one only tries several combinations whereas the second one tries all the possible combinations with the grid you provided.

We started with a random search to roughly evaluate a good combination of parameters. Once this is complete, we use the grid search to get more precise results.

**Randomized Search with Cross Validation**

import numpy as np

# Number of trees in random forest

n\_estimators = [int(x) for x in np.linspace(start = 10, stop = 1000, num = 11)]

# 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 = 5)]

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 = {'randomforestregressor\_\_n\_estimators': n\_estimators,

'randomforestregressor\_\_max\_features': max\_features,

'randomforestregressor\_\_max\_depth': max\_depth,

'randomforestregressor\_\_min\_samples\_split': min\_samples\_split,

'randomforestregressor\_\_min\_samples\_leaf': min\_samples\_leaf,

'randomforestregressor\_\_bootstrap': bootstrap}

# Use the random grid to search for best hyperparameters

from sklearn.model\_selection import RandomizedSearchCV

# Random search of parameters, using 2 fold cross validation,

# search across 100 different combinations, and use all available cores

rf\_random = RandomizedSearchCV(estimator = my\_pipeline\_RF,

param\_distributions = random\_grid,

n\_iter = 50, cv = 2, verbose=2,

random\_state = 42, n\_jobs = -1,

scoring = 'neg\_median\_absolute\_error')

# Fit our model

rf\_random.fit(train\_X, train\_y)

rf\_random.best\_params\_

{'randomforestregressor\_\_bootstrap': True,

'randomforestregressor\_\_max\_depth': 35,

'randomforestregressor\_\_max\_features': 'auto',

'randomforestregressor\_\_min\_samples\_leaf': 2,

'randomforestregressor\_\_min\_samples\_split': 5,

'randomforestregressor\_\_n\_estimators': 1000}

**Grid Search with Cross Validation**

from sklearn.model\_selection import GridSearchCV

# Create the parameter grid based on the results of random search

param\_grid = {

'randomforestregressor\_\_bootstrap': [True],

'randomforestregressor\_\_max\_depth': [30, 35, 40],

'randomforestregressor\_\_max\_features': ['auto'],

'randomforestregressor\_\_min\_samples\_leaf': [2],

'randomforestregressor\_\_min\_samples\_split': [4, 5, 6],

'randomforestregressor\_\_n\_estimators': [950, 1000, 1050]

}

# Instantiate the grid search model

grid\_search = GridSearchCV(estimator = my\_pipeline\_RF,

param\_grid = param\_grid,

cv = 3, n\_jobs = -1, verbose = 2,

scoring = 'neg\_median\_absolute\_error')

# Fit the grid search to the data

grid\_search.fit(train\_X, train\_y)

grid\_search.best\_params\_

{'randomforestregressor\_\_bootstrap': True,

'randomforestregressor\_\_max\_depth': 30,

'randomforestregressor\_\_max\_features': 'auto',

'randomforestregressor\_\_min\_samples\_leaf': 2,

'randomforestregressor\_\_min\_samples\_split': 4,

'randomforestregressor\_\_n\_estimators': 1050}

**Final Model**

# Create the pipeline (imputer + scaler + regressor)

my\_pipeline\_RF\_grid = make\_pipeline(Imputer(), StandardScaler(),

RandomForestRegressor(random\_state=42,

bootstrap = True,

max\_depth = 30,

max\_features = 'auto',

min\_samples\_leaf = 2,

min\_samples\_split = 4,

n\_estimators = 1050))

# Fit the model

my\_pipeline\_RF\_grid.fit(train\_X, train\_y)

evaluate\_model(my\_pipeline\_RF\_grid, test\_X, test\_y)

Median Absolute Error: 13.57

RMSE: 125.04

We get better results with the tuned model than with default hyperparameters, but the improvement of the median absolute error is not amazing. Maybe we will have better precision if we use another model.

**Visualisation of all models’ performance**

The tuned Random Forest and XGBoost gave the best results on the test set. Surprisingly, the Multi Layer Perceptron with default parameters gave the highest Median Absolute errors, and the tuned one did not even give better results than the default Random Forest. This is unusual, maybe the Multi Layer Perceptron needs more data to perform better, or it might need more tuning on important hyperparameters such as the hidden\_layer\_sizes.

**Conclusion**

In this post, we modelled Airbnb apartment prices using descriptive data from the Airbnb website. First, we preprocessed the data to remove any redundant features and reduce the sparsity of the data. Then we applied three different algorithms, initially with default parameters which we then tuned. In our results the tuned Random Forest and tuned XGBoost performed best.